- Replies
- Mentions
- Hashtags
- More stickers

February 2015, Second Update

Photo Editor and Passcode Lock

February '15 delivered the most powerful **photo editor** among messaging apps – featuring auto-enhance, crop and rotate functions in its initial version. To keep those photos and chats secure, users were given the option to lock their app with a **passcode**.

On Telegram since February 25, 2015

- Photo editor
- Local passcode lock for apps

February 2015, First Update

Shared Files and Fast Mute

This update introduced the **Files tab** to better view all documents shared in a particular chat (some media outlets mistakenly thought this was when we introduced sending files). Additionally, the update brought in **Mute** shortcuts, helping users disable notifications more quickly.

On Telegram since February 1, 2015

- Shared files
- Mute notifications

January 2015

Stickers Done Right

January '15 saw the first **sticker** on Telegram. Since then, starting a message with a **single emoji** brings up a list of emotionally corresponding **sticker suggestions**.

The first official stickers from Telegram formed the classic Great Minds set. Artists were invited to publish their work on the **free** and **open platform**.

On Telegram since January 2015

- Stickers
- Sticker suggestions by emoji
- Official 'Great Minds' stickers
- Open platform for sticker artists

2014

December 2014

Telegram.me, Changing Numbers, PFS and more

Users with usernames were offered **Telegram.me/username** links, which open their profile page in Telegram. This update also enabled **changing** the **phone number** of a Telegram account. **Single-column mode** was added for Telegram Desktop, making multitasking easier on smaller screens.

> *Fun fact: In 2016, Instagram will restrict users from adding Telegram.me (and Snapchat) links to their bios.*

On Telegram since December 2014

- Use telegram.me links
- Change your phone number
- Perfect Forward Secrecy
- Single-column mode for Telegram Desktop
- GIF and Image Search on Android

November 2014

There were two Telegram updates on November 19, 2014.

Hiding Last Seen Time – Done Right

This update expanded **Privacy Settings** to cover **Last Seen** status, allowing unprecedented flexibility with *'Always Share With'* and *'Never Share With'* exceptions for individual users.

This was also when **account self-destruction** was introduced. Telegram doesn't aim to accum[illegible] away your SIM card and it gets a new owner in 6 months, they won't access your chats.

Material Design on Android, Instant Search for Messages and more

November '14 also delivered **Instant Full-Text Search** to iOS and Android. Since then you can quickly find any message you ever sent or received on Telegram.

GIF search was first supported on iOS. The Android app got a massive redesign, consistent with **material guidelines**.

On Telegram since November 2014

- Granular privacy settings
- Who can see my Last Seen time
- Instant Full-Text Search
- Account self-destruction
- GIF Search for iOS
- Android 2.0 with Material Design

October 2014

Public usernames, smaller timers for Secret Chats, and more

October '14 marks the beginning of properly recorded history on Telegram. The first update ever to be described in a **blog post** brought **public usernames**, adding the ability to share your Telegram contact without disclosing your **phone number**.

Secret Chats got a major upgrade, with self-destructing media showing **blurred thumbnails** and starting the countdown only after they are first opened. This prevented them from disappearing before actually being viewed. This was also when **screenshot notifications** were first introduced.

Telegram for Android received several updates, adding support for **Android Wear** and **tablets**, as well as **video compression**.

On Telegram since October 2014

- Usernames
- Secret Chats 2.0
- Support for Android Wear
- Support for Android tablets
- Video compression on Android

September 2014

GIFs, iPad support and broadcast lists on iOS

Telegram has supported **iPads** since September '14. In this update, the iOS app learned to play **GIF animations** and added **cloud search** for messages, as well as adding **search by file name** to the Shared Documents section. iOS also caught up with Android, adding support for broadcast lists.

On Telegram since September 2014

- New app: iPad support
- Send GIFs on iOS
- Broadcast lists on iOS
- Large photos in chats
- Cloud search

August 2014

Broadcast lists were first introduced to Android in August '14, only to be replaced with channels a little over a year later.

On Telegram since August 2014

- Broadcast lists

June 2014

Multiple photo upload and more

Telegram for Android caught up with iOS, allowing users to **preview photos** before

Telegram for Android caught up with iOS, allowing users to **preview photos** before sending — as well as select and send many photos at once. Notifications were improved to be more reliable.

On Telegram since June 2014

- Send multiple photos at once
- Preview photos before sending
- Improved notifications

March 2014

Voice Messages, Delete Messages for both sides in Secret Chats, Language Settings on Android

The March '14 update introduced **voice messages** to Telegram. Messages deleted in Secret Chats began disappearing for **both sides**.

Telegram for Android added the option to change the **app language** in Settings. The first languages to be added were German and Italian.

On Telegram since March 2014

- Voice messages
- Language settings on Android
- Message deletion for both sides in Secret Chats

February 2014

New apps, autodownload settings and contact management tools for Android

More Telegram apps entered the fray. An unofficial Telegram app for **Web** was introduced and later moved to web.telegram.org to become the **official web version**. Telegram for **Windows Phone** was created in a contest for app developers.

Meanwhile, an update introduced **autodownload settings** and new tools to manage contacts on Android.

On Telegram since February 2014

- New app: Stand-alone Web version
- Autodownload settings
- Contact management tools
- Windows Phone support

January 2014

Send documents and files

First thing in 2014, Telegram allowed **documents of any type** to be sent, including **.pdf**, **.doc**, **.png**, **.mp3**, etc.

An unofficial Telegram app for PC was announced, which would later become the official **Telegram Desktop**.

On Telegram since January 2014

- New app: Stand-alone Desktop app
- Send documents on iOS and Android
- Send documents on Telegram Desktop

2013

December 2013

Crowdsourcing a more secure future

The only significant vulnerability to ever be discovered in the MTProto protocol was **fixed** during the First Telegram Crypto Contest. The researcher who discovered it was awarded a bounty of **$100,000**.

On Telegram since December 2013

- Bug bounty program
- Detailed technical FAQ

October 2013

After a contest for Android developers, the alpha version of **Telegram for Android** was officially launched. Telegram apps got **Secret Chats** with **self-destruct timers**.

The MTProto Protocol specification and Telegram API were **fully documented** and the code of the apps became **open source**.

On Telegram since October 2013

- **End-to-end encrypted chats**
- **Self-destruct timers**
- **Open source apps**
- **Documented protocol and API**

August 14, 2013

Telegram for iOS was launched.

Congratulations on completing this (not very) brief course on the history of Telegram. You just got an honorary PhD in Instant Messaging – unless you simply jumped down here, in which case you still have a long way 👆 to go to get back to the top.

Armed with this much knowledge, you should probably join our Volunteer Support Force. And if that's not your cup of tea – just tell us in the comments which feature you didn't know existed.

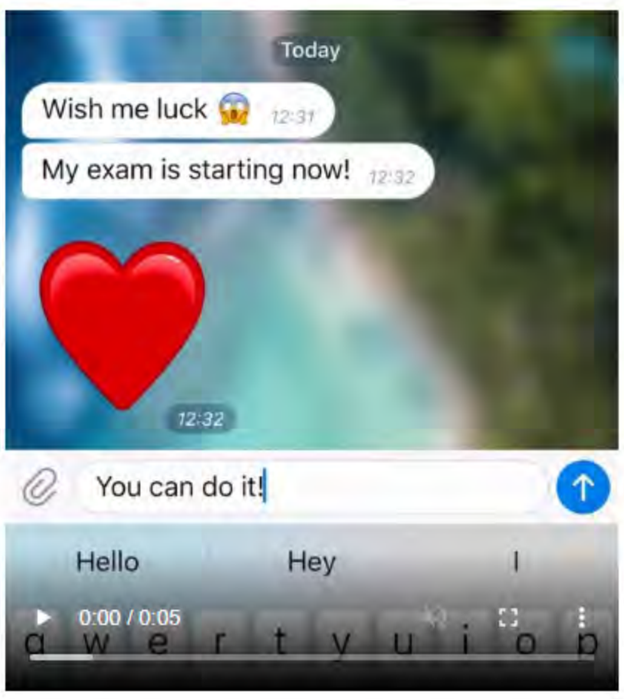
August 15, 2019
The Telegram Team

Forward Tweet

Silent Messages, Slow Mode, Admin Titles and More

The previous update brought more movement to Telegram – this one brings more peace of mind. You can now message friends freely when you know they are sleeping, studying or attending a meeting.

Simply **hold** the Send button to have any message or media delivered **without sound**.

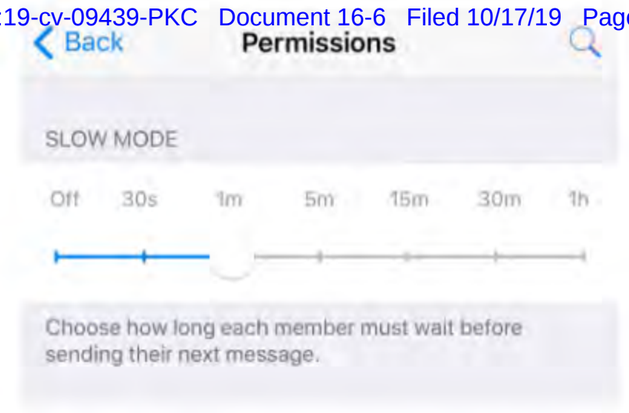


Your recipient will get a **notification** as usual, but their phone won't make a sound – even if they forgot to enable the *Do Not Disturb* mode.

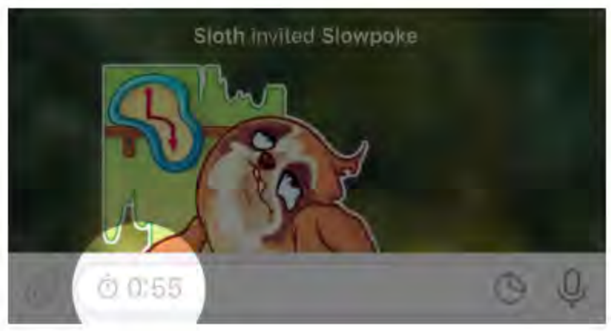
This also works in **groups**, should you get an urgent idea at five in the morning – but not urgent enough to wake up everyone in your work chat.

Slow Mode

In case a group you manage is getting hard to follow, the Group Permissions section now features a **Slow Mode** switch.



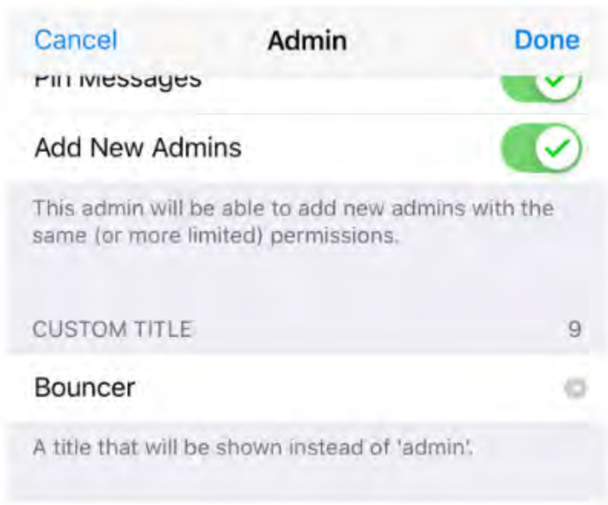
When an admin enables Slow Mode in a group, you will only be able to send **one message** per the interval they choose. A timer will show how long you have to wait before sending your next message.



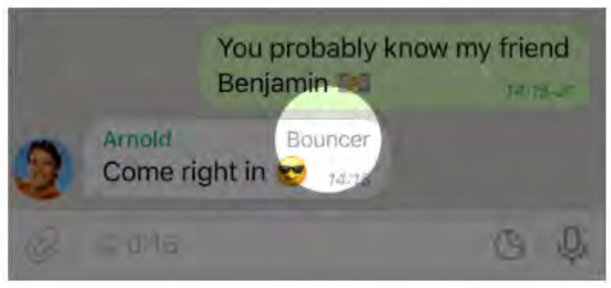
Slow Mode can make conversations in the group more orderly, while raising the value of each individual message. Keep it on permanently, or toggle as necessary to throttle rush hour traffic.

Admin Titles

If new time-lord powers aren't enough, group owners can now set **custom titles** for admins like 'Meme Queen', 'Spam Hammer' or 'El Duderino'.



As with the default admin labels, custom titles are shown with every message in the group so members know that they're talking to the designated 'Myth Buster'.

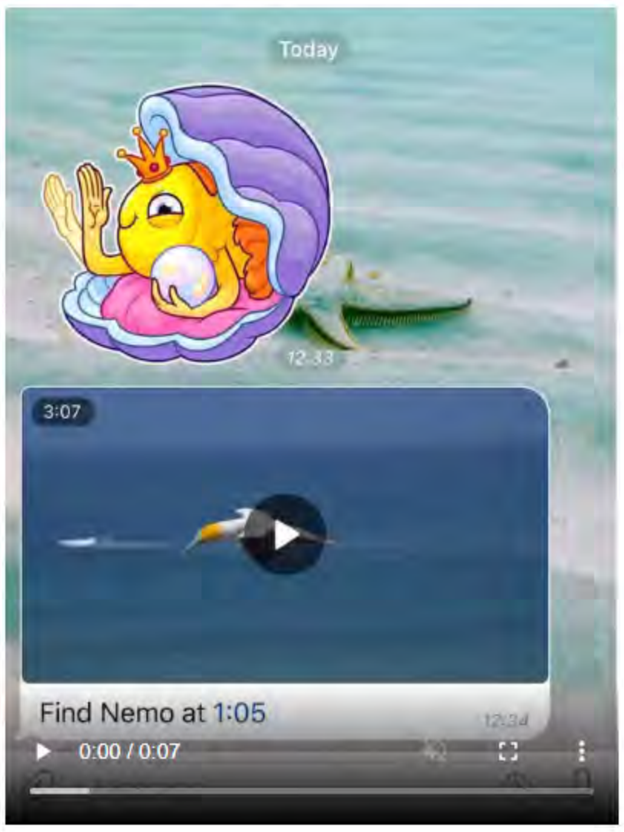


To add a custom title, edit the admin's rights in Group Settings.

Timestamps and Improved Scrubbing

Videos now display **thumbnail previews** as you scrub through, to help you find the moment you were looking for.

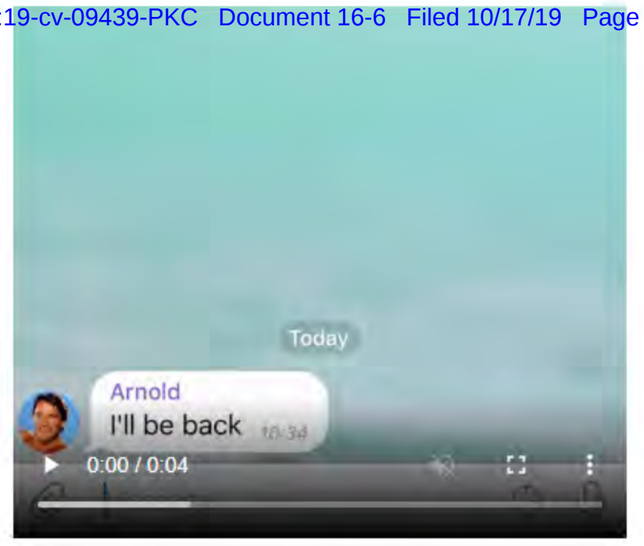
If you add a timestamp like **0:45** to a **video caption**, it will be automatically highlighted as a link. Tapping on a timestamp will play the video from the right spot. This also works if you mention a timestamp when **replying** to a video.



Timestamps are also supported for **YouTube** videos, in case you want to mark your favorite moments when sharing a Kurzgesagt episode.

Animated Emoji

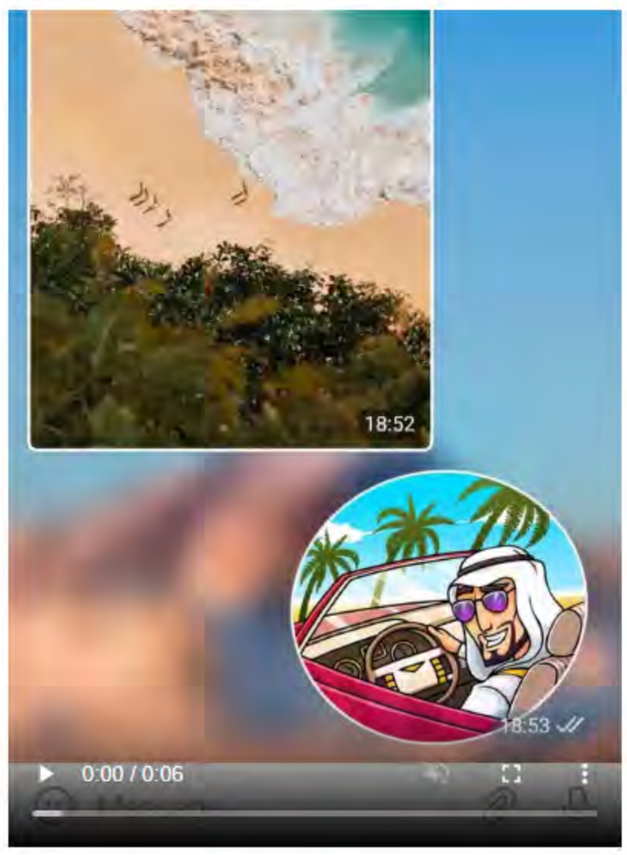
When you have animated stickers, why not go one step further and get **animated emoji**? To check them out, send a single ❤️, 👍, 😔, 😳 or 🥳 to any chat.



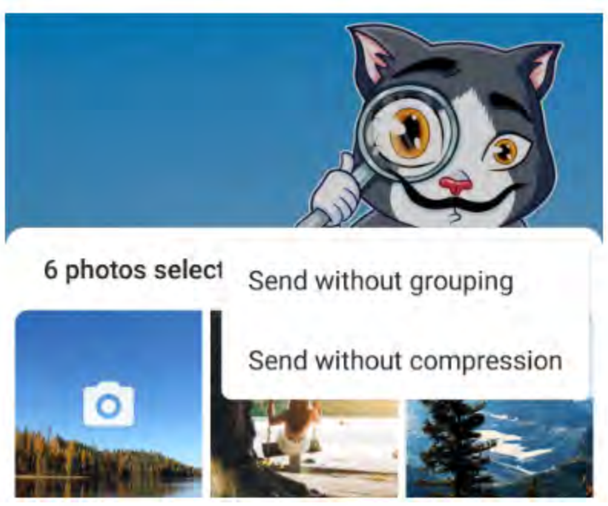
If your life feels a little too animated recently, Sticker Settings now offer a toggle for **looped playback**. When disabled, animated stickers will play just once then stay still.

Android's New Attachment Menu

Android's attachment menu got a makeover, giving media more real estate. **Larger thumbnails** make it easier to pick photos and videos at a glance, and swiping up will reveal your full **Gallery** for better browsing.



You can scroll left and right through the other attachment options like locations, polls and music. When selecting media, tap '...' to send items as uncompressed documents.



Accent Colors for Night Mode on iOS

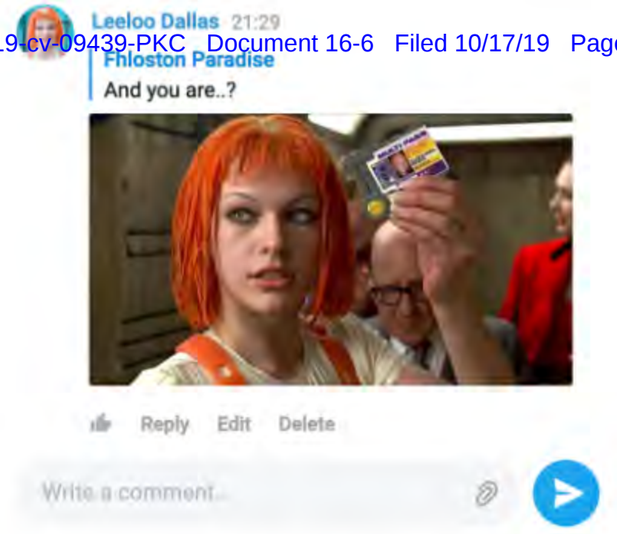
iOS users can now choose **accent colors** for **night themes**. The night doesn't always have to be black and blue, after all.

Comments Widget

Comments.App, our tool for commenting on **channel posts**, now lets you add a **comments widget** to your **website**.

With the widget in place, Telegram users will be able to log in with just two taps and **leave comments** with text and **photos**, as well as **like**, **dislike** and **reply** to comments from others.

They can also **subscribe** to comments and get notifications from @DiscussBot.



Open this page in your browser to try the new widget – it doesn't support Instant View pages... yet. 😈

That's all for now, and don't worry — the next Telegram update won't be sent silently.

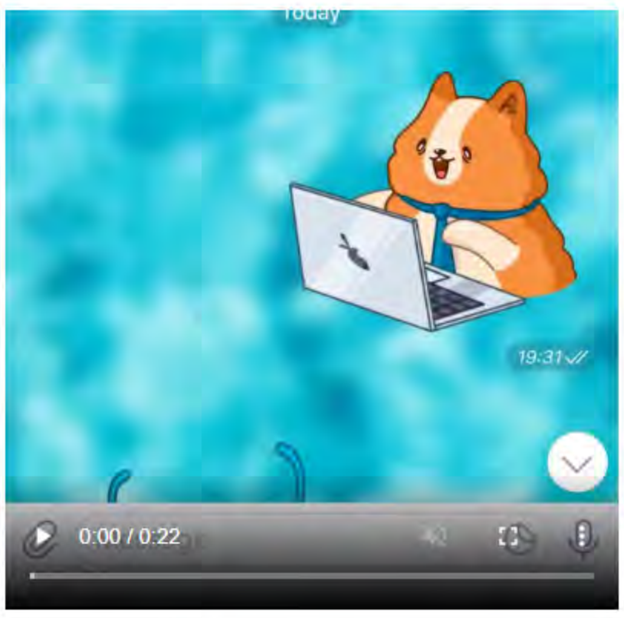
August 9, 2019
The Telegram Team

Forward Tweet

Animated Stickers Done Right

We launched stickers back in January **2015**. Since then, the Telegram sticker format has been adopted by other apps to reach a total of **2 billion** people. Today we're introducing a **new format** for **animated stickers**.

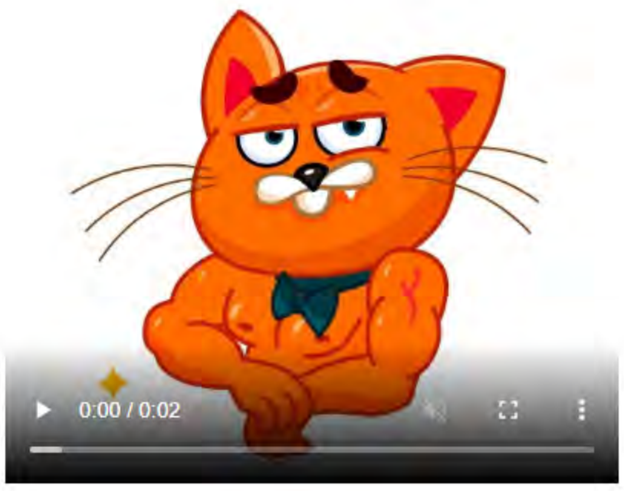
We asked ourselves: Can animated stickers have **higher quality** than static ones while taking **less** bandwidth? The answer turned out to be *YES* (but only after we told developers they'd get moving cat pictures).



Smooth Animations, Tiny Size

Telegram engineers experimented with vector graphics, packaging methods and forbidden magic to create the Lottie-based **.TGS** format, in which each sticker takes up about **20–30 Kilobytes** – **six times** smaller than the average photo.

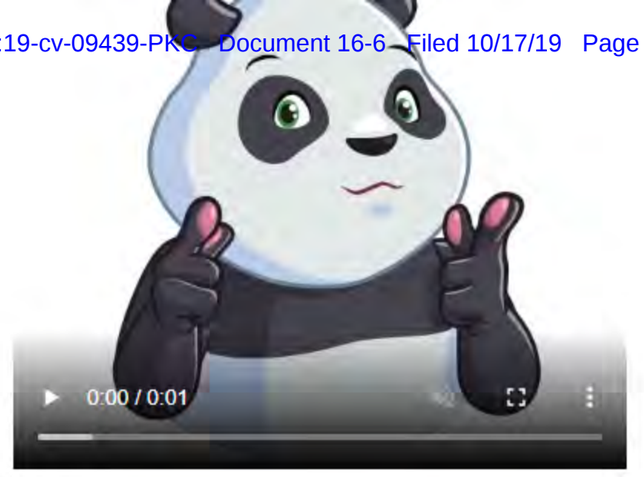
Thanks to various optimizations, animated stickers consume **less battery** than GIFs and run at a smooth **60 frames per second**. If a picture is worth a thousand words, that's **180,000** words per sticker.



Open Platform

Naturally, animated stickers are a **free platform**. All artists are welcome to **create** new sets and **share** them with Telegram users.

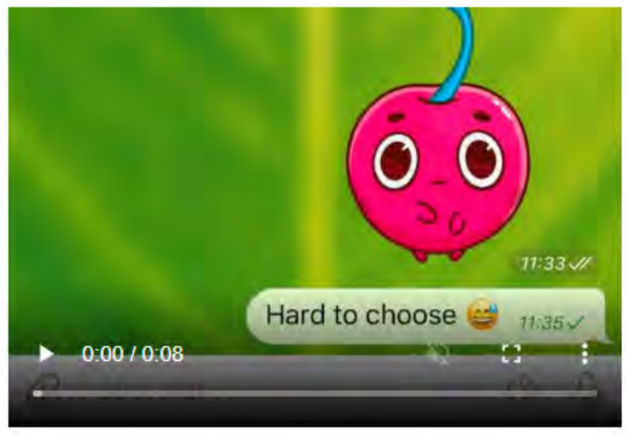
Like its static predecessor, the Telegram animated sticker format is likely to become the new industry standard in messaging. Check out this quick guide to get started.



Starter Packs

To get your conversations moving right away, our artists have created a few sample sets ranging from Rambunctious Rodents to Sentient Snacks. You can find more animated sticker sets in the 'Trending' section of your sticker panel. 🔥

As always, the fastest way to find a sticker that fits your mood is to type in a **relevant emoji** – Telegram will immediately suggest matching stickers.



Keep an eye out for new animated stickers – and our next update.

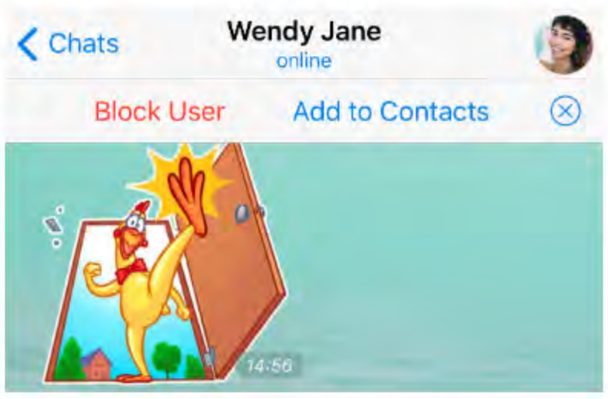
July 6, 2019
The Telegram Team

Forward Tweet

Location-Based Chats, Adding Contacts Without Phone Numbers and More

In the previous update, we improved privacy settings and added a way for you to control who can see your phone number. Today we're making it easier to **exchange contact info** on Telegram.

All new chats now have an **Add to Contacts** button at the top. This allows you to quickly add **anyone** who messages you to your Telegram contacts, even if you don't know their phone number yet.

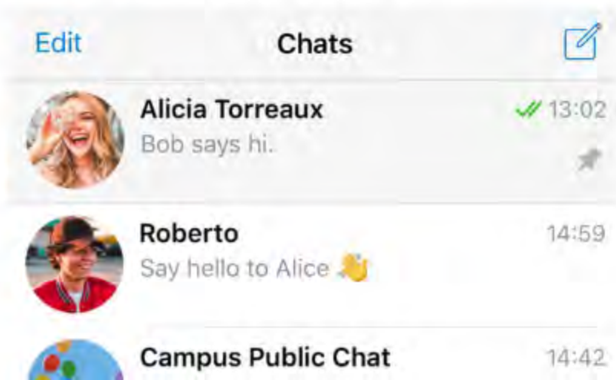


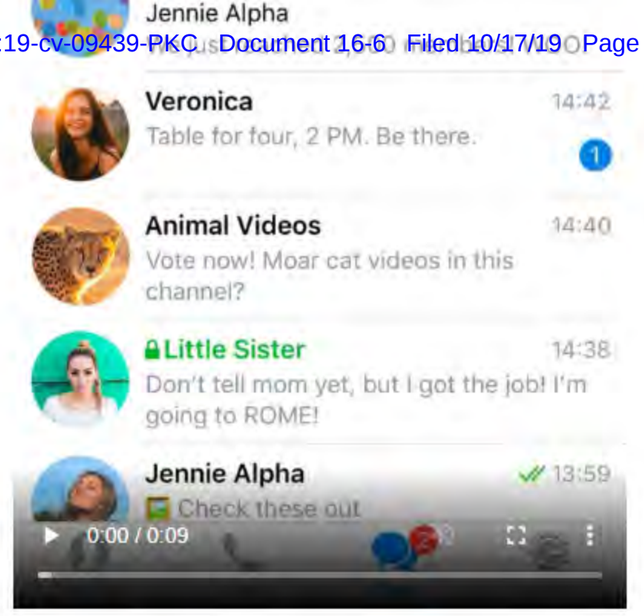
If you'd prefer they disappeared instead, the **Block** button is right next door.

Add People Nearby

Ever scrambled for business cards at a beach party? Or dropped somebody's phone into the pool in a "let me type in my number for you" moment? Worry no more.

Simply open *Contacts > Add People Nearby* to quickly **exchange contact info** with Telegram users who are **standing next to you** (and also have this section open).

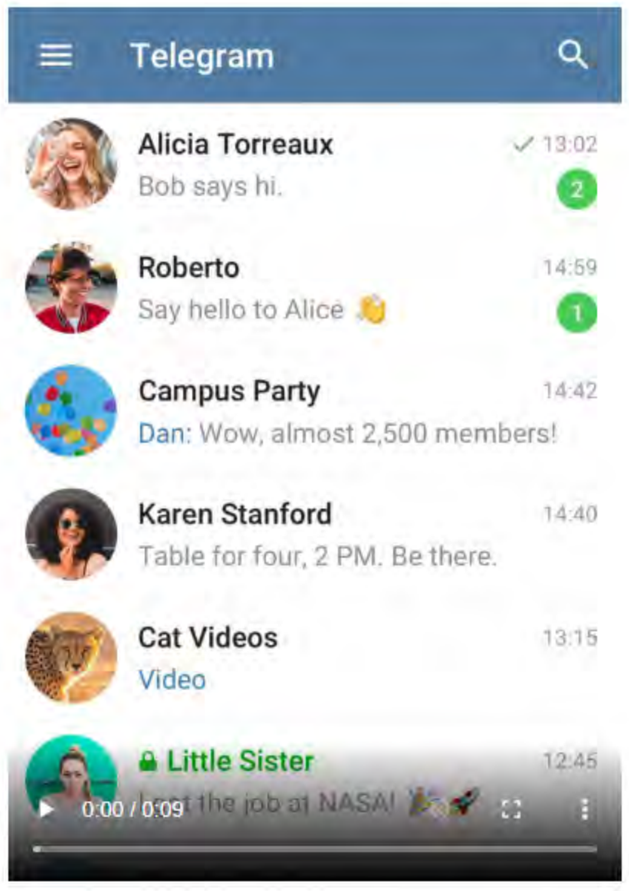




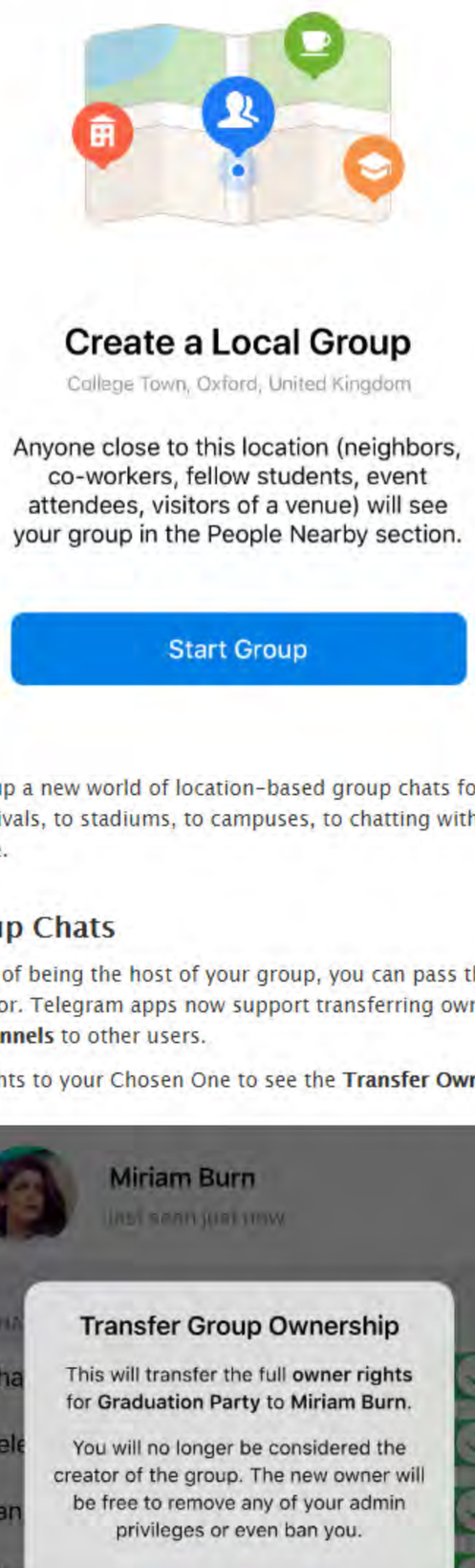
This feature comes in especially handy when several people meet to perform the take-my-number dance. Now you can catch all your pokémon in just a few taps.

Location-Based Chats

Speaking of pokémon, the new People Nearby section also shows **Groups Nearby** – location-based group chats open for anyone around to join.



Tap **Create a Local Group** to unite your dormitory or apartment building, and maybe you can get Todd in 2C to finally turn his music down.

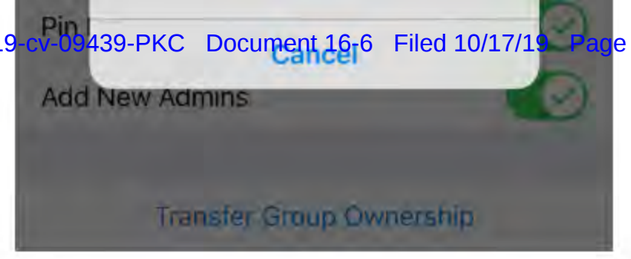


This update opens up a new world of location-based group chats for anything from conferences, to festivals, to stadiums, to campuses, to chatting with people hanging out in the same cafe.

Transfer Group Chats

If you ever get tired of being the host of your group, you can pass the burden on to another administrator. Telegram apps now support transferring ownership rights from any **groups** and **channels** to other users.

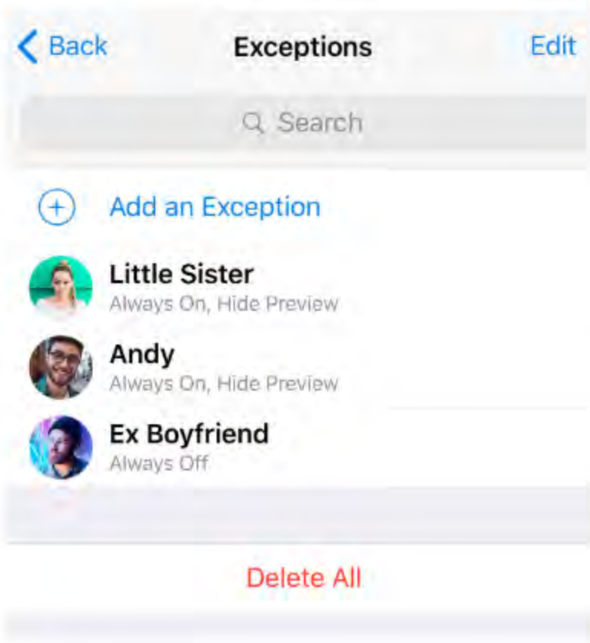
Grant full admin rights to your Chosen One to see the **Transfer Ownership** button.



Whether your watch has ended, or you have some business to attend to in King's Landing, passing the torch is a simple, two-tap affair.

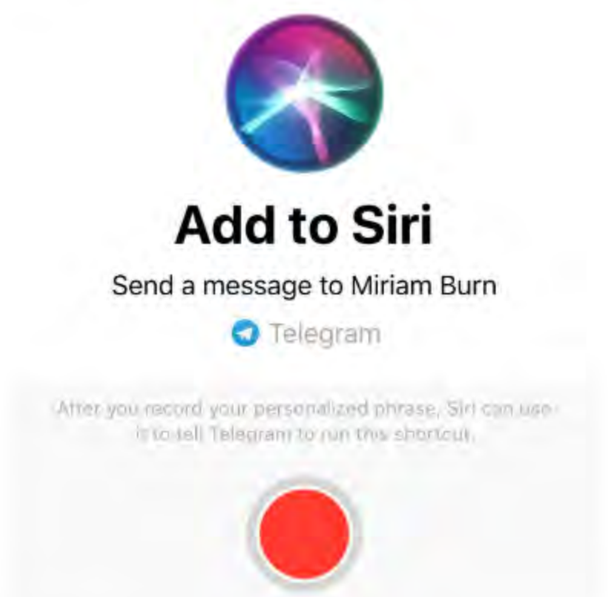
Enhanced Notification Exceptions

Notification Exceptions just got more powerful. You can now toggle **message previews** for specific chats. If you have many exceptions, use **Search** to find the right chat — or **'Delete All'** to get back to square one instantly.



Siri Shortcuts

Owners of iOS devices can now use Siri shortcuts to open chats with people. No hands just got no-handsomer.

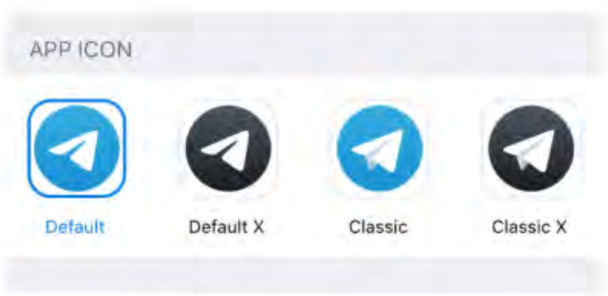


Theme Picker and Icon style

We've also revamped the **Appearance** settings on iOS so that it's easier to see what the different themes will look like even before you apply them.

A while back, the Telegram X app for **iOS** was promoted, becoming the **official** Telegram, and its previous shell was removed from the store. If you are still using it for some reason, this is a good time to switch — ol' Telegram X is now three versions behind the main branch.

Some users told us they didn't want to switch because they liked the Telegram X icon better. So this update adds a way for your iOS app icon to get back in black:



By the way, Telegram X for **Android** is still there and is not going anywhere for now. If you're looking for an alternative interface, feel free to give it a try.

June 23, 2019
The Telegram Team

Forward Tweet

Focused Privacy, Discussion Groups, Seamless Web Bots and More

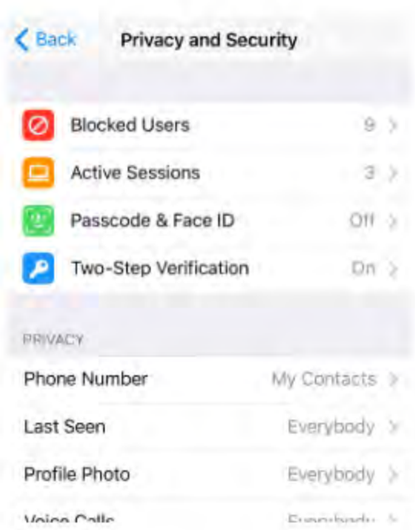
Telegram is about privacy. In **2014** we pioneered granular privacy settings in messaging. Today we are making them even more flexible with **exceptions for group chats**.

From now on, you can make something visible for all your classmates in one group chat and keep it private from, say, all your colleagues in another – with just two taps:

Settings will **adjust automatically** as people join and leave the groups. So when your half-brother unexpectedly gets a job as a data broker, you'll just need to kick him out of one group to update all your settings.

Who can see my phone number?

On Telegram, you can send messages in private chats and groups without making your phone number visible. But there may be cases when you want to make your number known (e.g. to all your coworkers), so we added a new dedicated privacy control – **Who Can See My Number**.



Redesigned privacy settings on iOS

Meet Seamless Web Bots

We've made it easier to **integrate bots with web services**. Bots can now help you **log in** with your Telegram account on a website when you open a link. If you allow them to, you'll be logged in by the time the page loads in the browser:

While this is purely optional, it opens the door for a vast variety of new bots. To try out this seamless authorization, press the 'comments' button under this post.

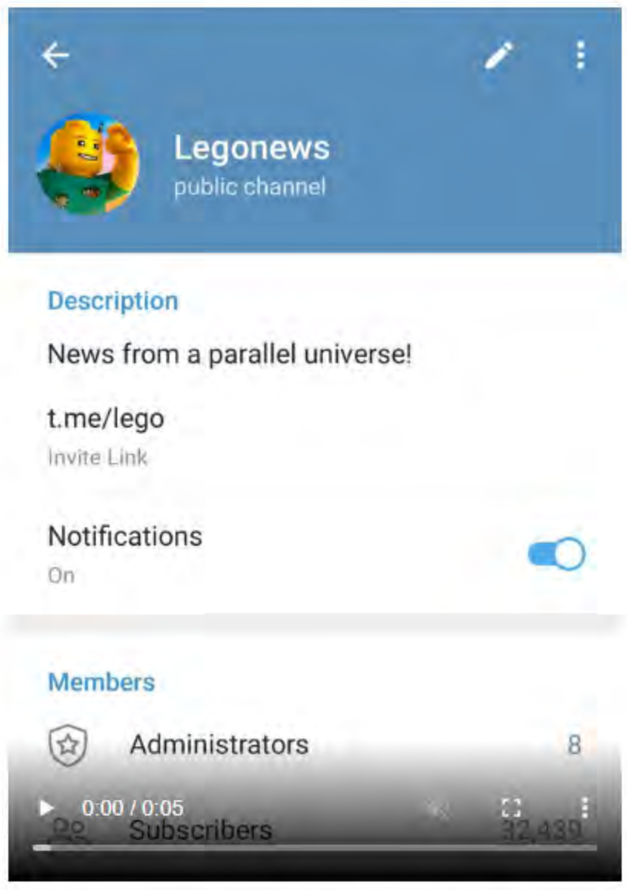
You can also make our sample @discussbot an admin in any of your broadcast channels to get a comments button under the posts you publish. The comments button opens a website where you are already logged in and ready to leave a comment. The bot will notify you if someone replies to what you wrote there.

Anyone can create similar bots to connect their existing services to Telegram on the fly. Integrating all kinds of social, gaming, productivity, dating or e-commerce services into your channels is now a breeze.

Broadcasts meet Group Chats

Telegram channels are a tool for broadcasting your thoughts to unlimited audiences. Telegram group chats offer a democratic way for communities of up to **200,000** members to discuss things.

Ever since we launched channels and groups, users have been asking us to add **discussions to channels** and **announcements to groups**. With this update, admins can add a group chat to their channel to serve as a discussion board:



Subscribers will see a **'Discuss'** link on the bottom panel, and each new post from the channel will be automatically forwarded to and **pinned** in the discussion group.

View public channels

Speaking of channels, you can now view **any public channel** from the web – even if you aren't logged in to Telegram. The same also goes for those retro people who don't have a Telegram account at all. Yet.

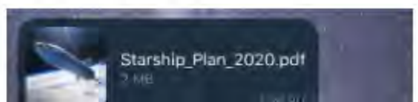
Simply open the channel link in a browser and select "Preview channel" to see something like this: t.me/s/telegram

Scam Alerts

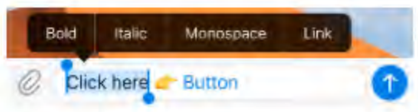
Telegram apps will now show a **scam** label for suspicious accounts.

iOS Goodies

In other news, Telegram 5.7 for iOS introduces **thumbnails for PDF** files. Keep in mind that Telegram lets you share files of any type, up to **1,5GB** each in size (so you can telegram a PDF payload worthy of Elon's rockets).



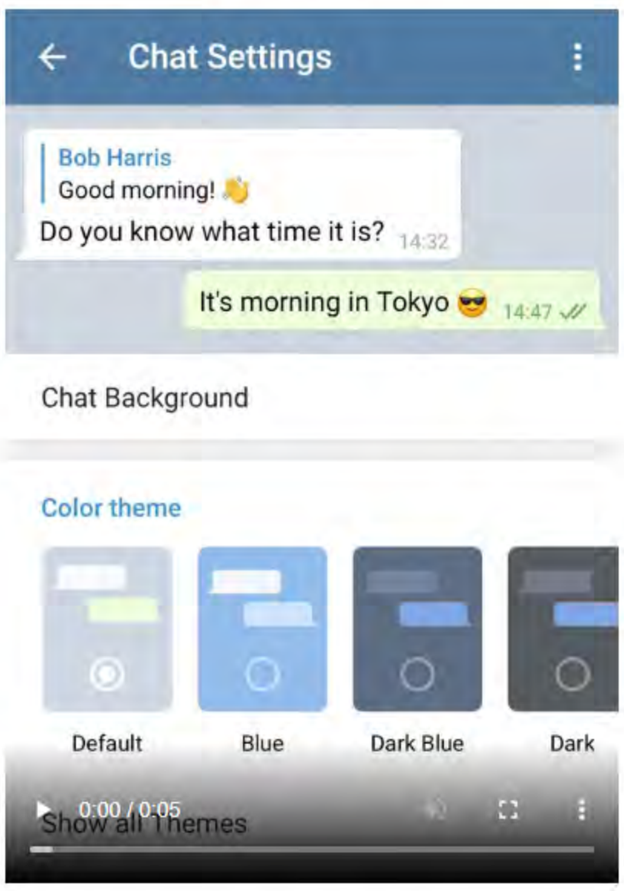
Telegram for iOS also catches up with the rest of our apps in terms of **text links**. You can now make any text a link to a website, keeping all the cords under the carpet.



Mind that people will get a warning about where exactly the link leads when they open it. (Hint: use URL shorteners if you want to rick-roll people).

Android Delights

As for Android, we've redesigned the majority of confirmation dialogs in the app, and improved the **design** for message search and adding people to groups. Additionally, the app got a new **theme switcher** in *Chat Settings*.



Have fun with all that and stay tuned for our next updates.

May 31, 2019
The Telegram Team

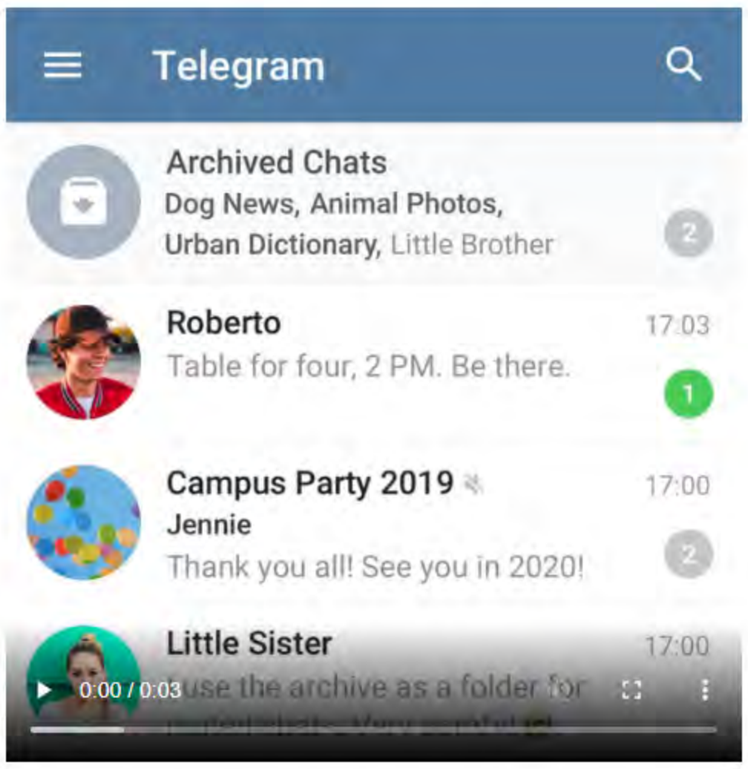
Forward Tweet

Archived Chats, a New Design and More

Today's update gives you the tools to sort your messages with **archived chats**, a **new design** on Android and a handful of other nifty features to make the most of your messenger.

Everything in its place

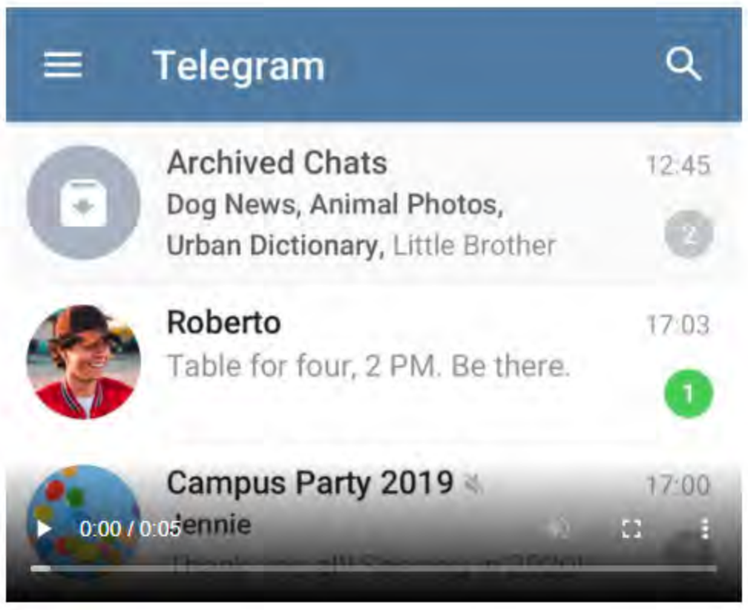
Introducing **archived chats**, the new tool for spring cleaning in your chat list. Sort your active and inactive chats, separate personal stuff from work or banish annoying contacts to your archive for some spectacular revenge!



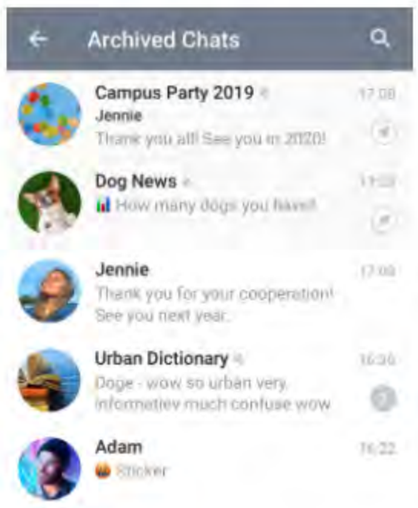
Swipe left on a chat to transfer it to your **archived chats** folder. When an archived chat

gets a notification, it will pop out of the folder and back into your chat list. Muted chats will stay archived forever.

You can **hide the archive** by swiping left on it. See it again by dragging the screen down.

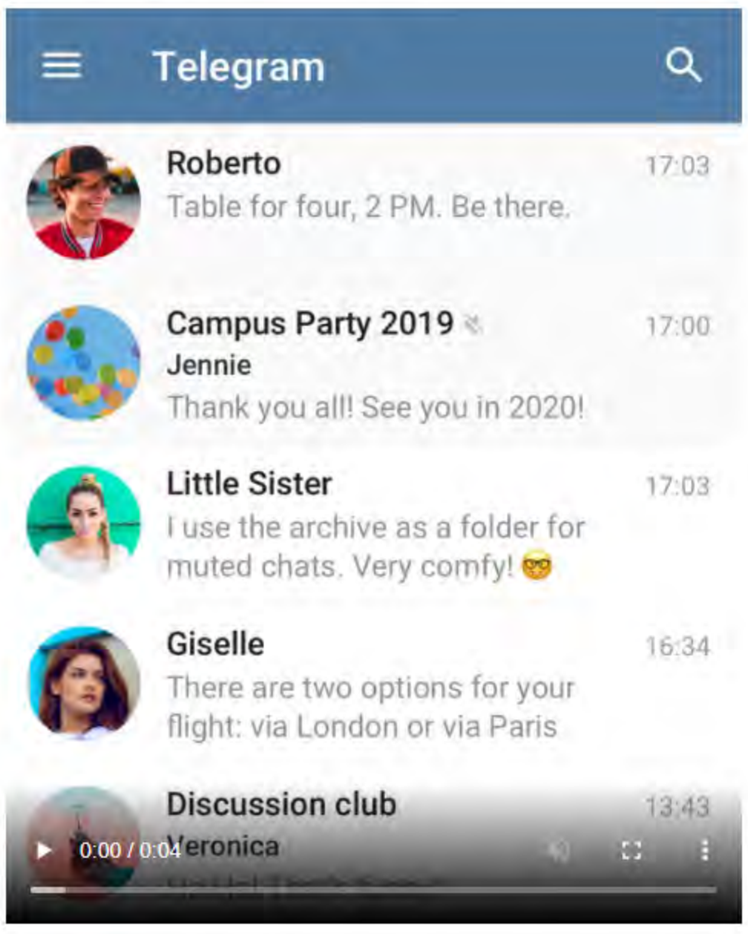


Pin an **unlimited** number of chats in your archive to keep your messages in the order you want. When chats pop out of your archive with a notification, you can archive them again to return them to their original place within the folder.



Bulk actions for a busy chat list

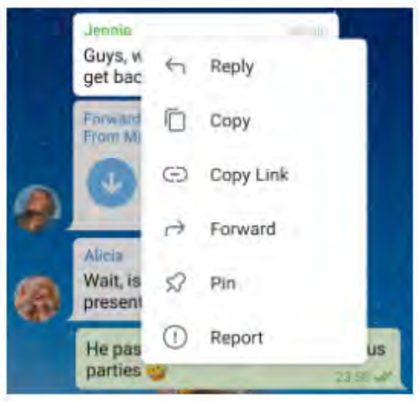
Time is money and money buys cat food, so don't waste time and make the most of the bulk actions in your chat list, now also available on Telegram for **Android**. Long tap on a chat to open the new menu where you can **select multiple chats** and then pin, mute, archive or delete them, all faster than ever before.



Android's new clothes

Telegram for Android got a lot slicker, starting with the app's **new icon** and down to every menu in the app sporting a **new design**.

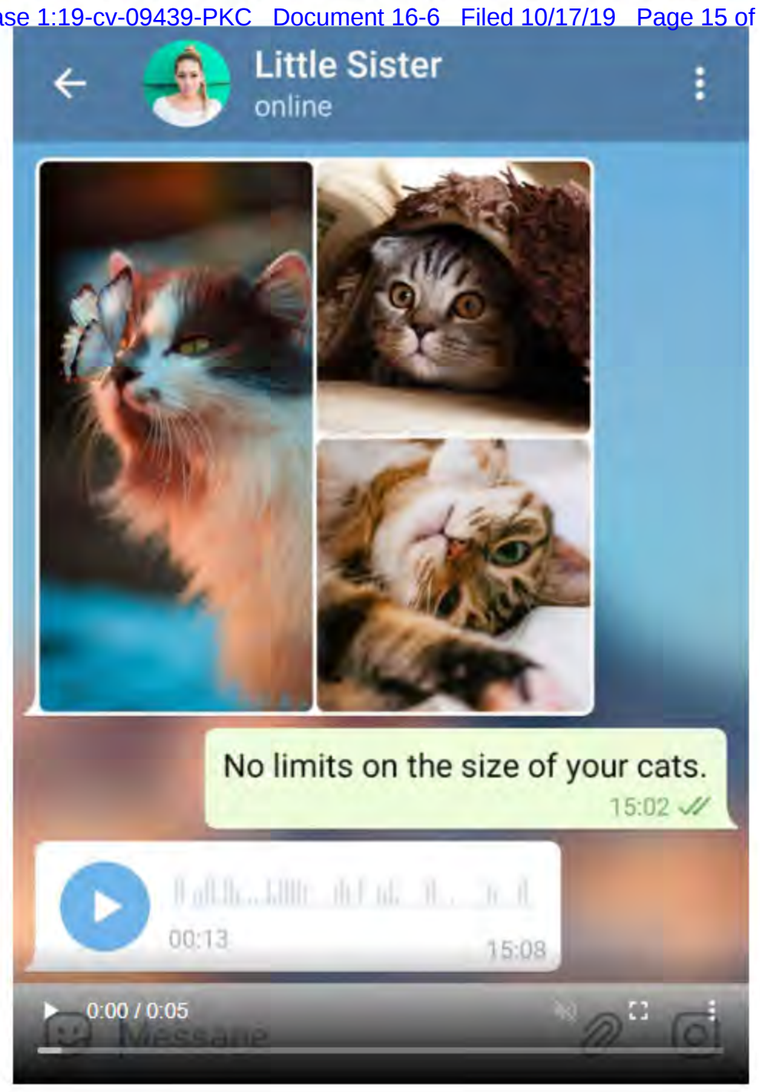
New app icon



Slick new menus

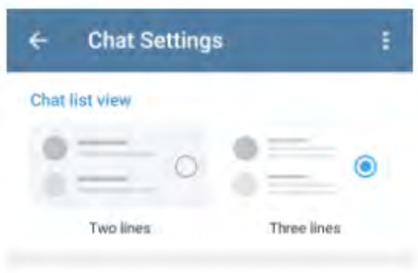
Selecting messages in chats is now not only more stylish but also more functional –

note the new **quick forwarding button** which appears right under your thumb:



More info at a glance

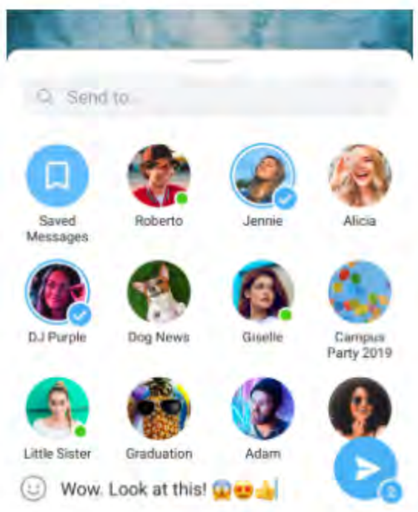
With the **expanded chat list**, you can see more text from the messages in your list. Simply select **Three lines** in the *Chat Settings* menu to see up to three lines per chat instead of the usual two.



Chat list view settings

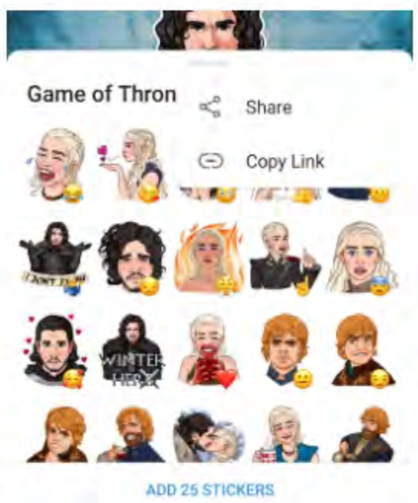
Share and share a lot

Want to share a message far and wide? Pull up the new streamlined **sharing menu** to cover the entire screen and select your whole gang. And don't fret if you're short on words, the comment field now supports **emoji**.



New sharing menu

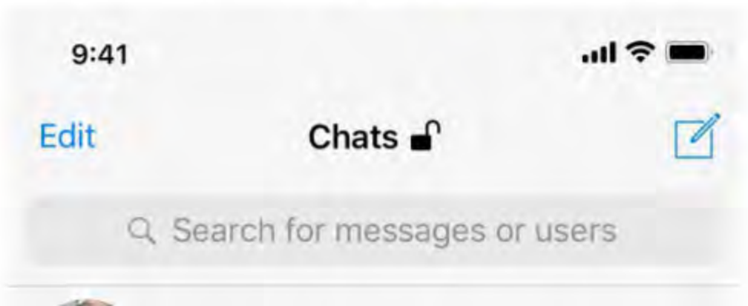
The new design also makes it easier to share **sticker packs**.

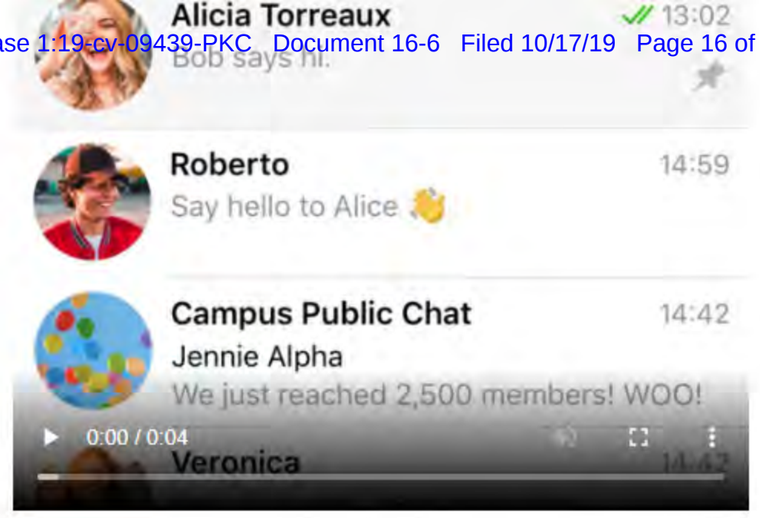


New sticker pack viewer

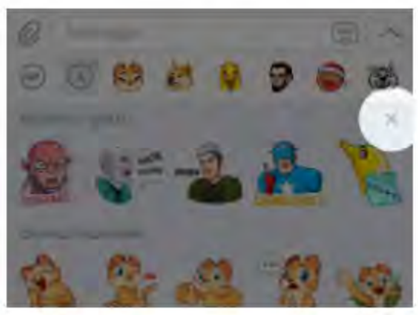
You shall not pass

Meanwhile on iOS, **passcode** settings have been made more stylish and more robust to accommodate **6-digit codes**, in addition to the previous 4-digit and custom alphanumeric options. Keep it secret, keep it safe 🔒.





Another new feature on iOS allows **clearing** your **recently used stickers** so nobody can prove how much you love sending funny dog stickers.



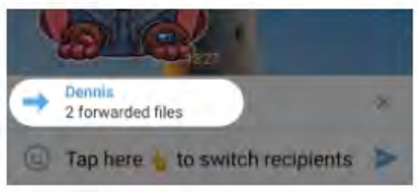
Clear recent stickers

We also thought that **large emoji** feel more natural without chat bubbles and made them look like little stickers instead. Aren't they cute?

New look for large emoji

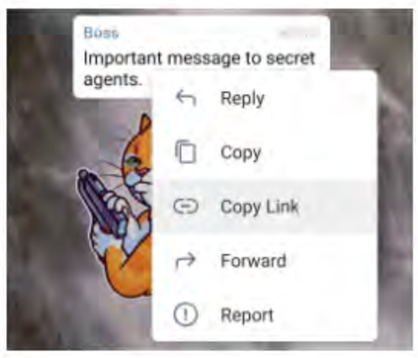
Easier forwarding, links to messages, online badges

Ever selected the wrong chat when **forwarding a message**? Tap the message snippet above the text box before sending to **change** where the message is forwarded to.



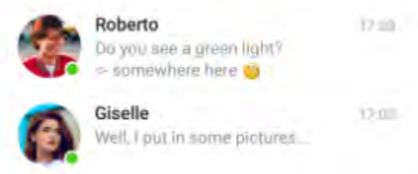
Switch recipient

If you'd rather point to a particular spot in a conversation, you can now **copy links** to messages in **private** groups and channels – just like you could with public messages. Needless to say, links to private chats will only work for members of their respective communities.



Links to individual messages

Last but not least, you can now instantly see who's **online** from the **chat list** and **sharing menu** to find out who else might be watching cat videos in the middle of the night.



As always, stay tuned for more updates on all our platforms — we look forward to popping back out of your archive with the next version.

May 9, 2019
The Telegram Team

Forward Tweet

Taking Back Our Right to Privacy

For us, your private data is **sacred**. We never use your data to target ads. We never disclose your data to third parties. We store only what is absolutely necessary for Telegram to work.

In **2013**, we gave millions of people power over their data with end-to-end encryption.

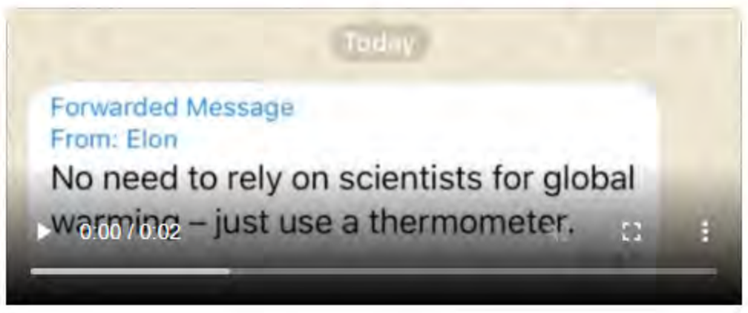
Today, we are giving hundreds of millions of users **complete control** of any private conversation they have ever had. You can now choose to delete any message you have sent or received from both sides in **any** private chat. The messages will disappear for both you and the other person – without leaving a trace.

Unsend Anything

The "Unsend" feature we introduced 2 years ago worked only for messages sent by you and only for 48 hours. Now you can "unsend" messages you have **received** as well, and there is **no time limit**. You can also delete any private chat entirely from both your and the other person's device with just two taps.

Anonymous Forwarding

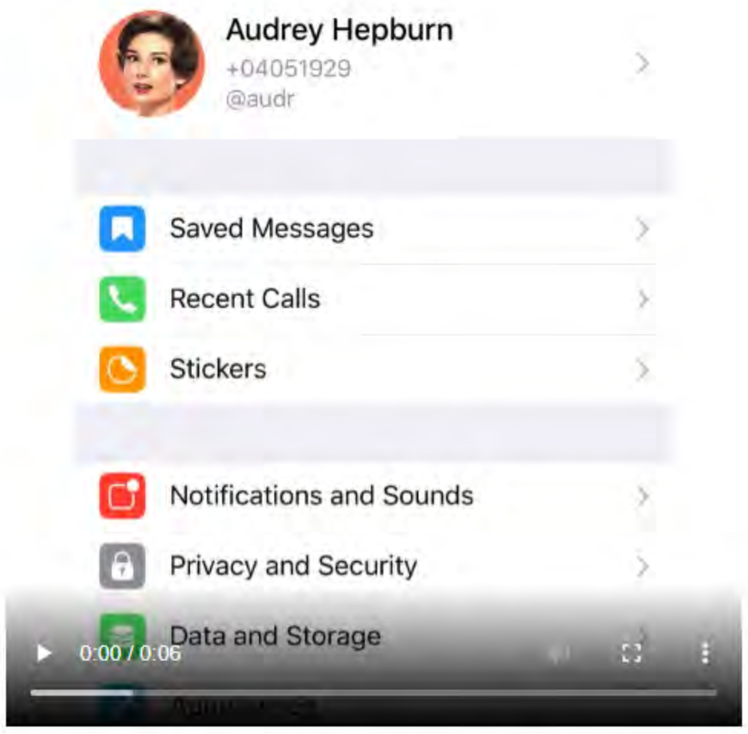
To make your privacy complete, we've also introduced a way to restrict who can **forward** your messages. When this setting is enabled, your forwarded messages will no longer lead back to your account — they'll just display an unclickable name in the "from" field. This way people you chat with will have no verifiable proof you ever sent them anything.



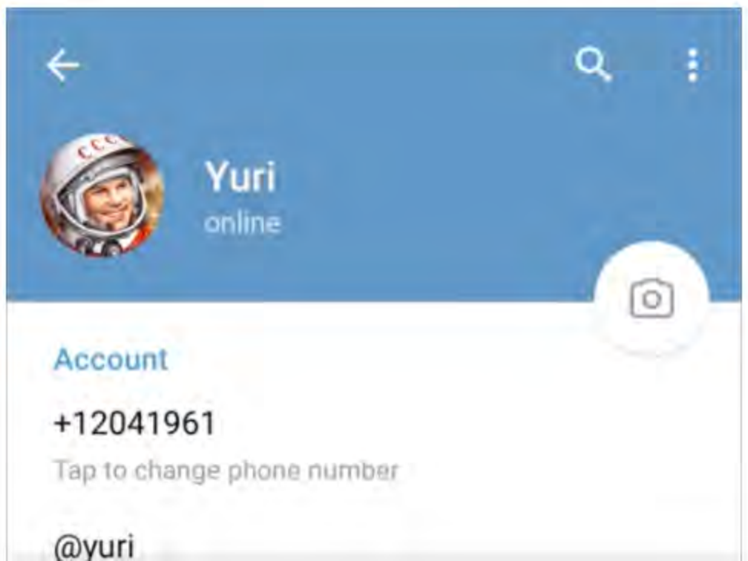
Look for "Forwarded messages" in *Privacy and Security* settings. By the way, you can now also restrict who can view your profile photos.

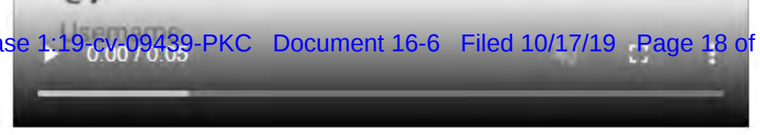
Settings Search

Since the Settings section keeps getting bigger, we've added a **search tool** that allows you to quickly find any **settings** you need.



This new search tool in Settings also shows answers to any Telegram-related questions based on the **FAQ**:

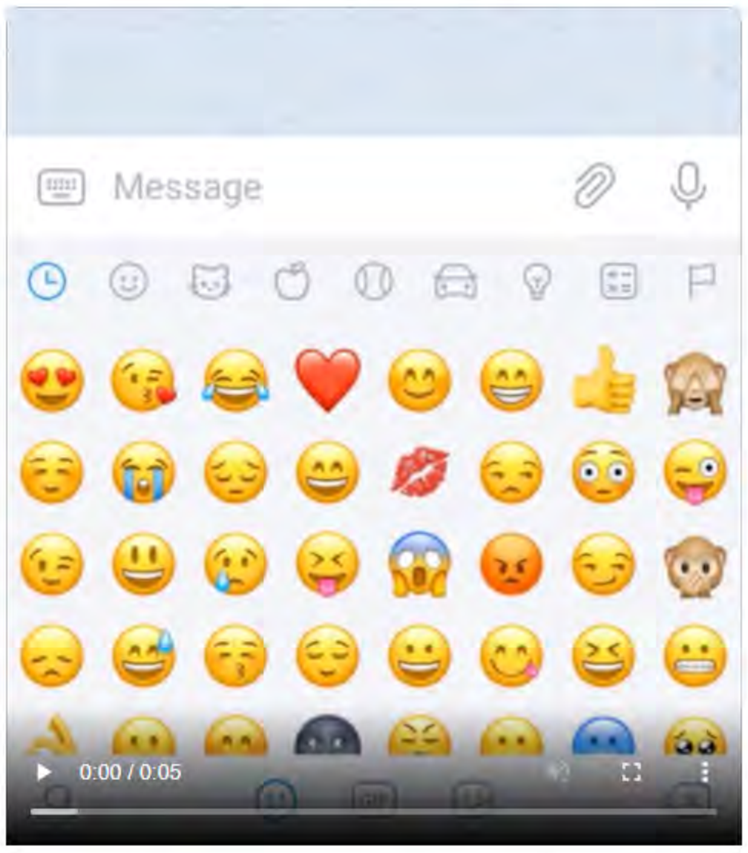




Emoji Search and GIFs

The GIF and stickers search has been upgraded on all mobile platforms – it now looks better and finds more cats. Any GIF you find can be previewed by tapping and holding. Sticker packs now have **icons**, which makes selecting the right pack easier. Large GIFs and video messages on Telegram are now streamed, so you can start watching them without waiting for the download to complete.

On Telegram for **Android**, you can now use keywords in many languages to find any **emoji**. If we are missing a keyword for an emoji, you're welcome to suggest it here – emoji search will constantly be improving without the need to update Telegram.



You'll also see a list of related emoji when typing a message. If you like an emoji enough to send it without any accompanying text, the emoji will appear larger in the chat on Android (iOS coming soon).

VoiceOver and TalkBack

We've added support for accessibility features – VoiceOver on iOS and TalkBack on Android. These gesture-based technologies give you spoken feedback so that you can use Telegram without seeing the screen.

> Tell us what you think the next update should be.

March 24, 2019
The Telegram Team

Forward Tweet

Autoplaying Videos, Automatic Downloads and Multiple Accounts

Today's update will make Telegram chats livelier with **Autoplaying Videos**. Smaller videos will start playing without sound when they reach your screen. To unmute them, simply press the volume buttons on your device.

If you like to be in control of your data usage, try the new **auto-download settings**. It's easy to see your current settings at a glance and we've added a new quick way to switch between *Low*, *Medium* and *High* presets for Mobile, Roaming and Wi-Fi.

You can also manually set up automatic downloads by chat type, media type and file size. The app will remember your choices as the **Custom** preset in case you need to temporarily switch to *Low* and back — or the other way around.

To Each According to Their Needs

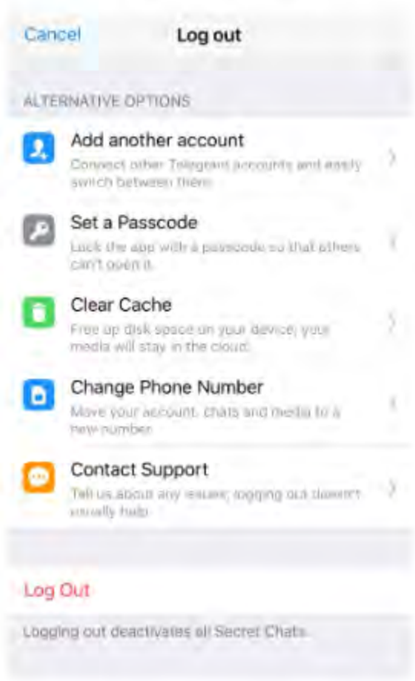
Default settings for data usage have become more generous but depend on the affordability of mobile data in your country. We know that in some places it's easier to buy an aircraft carrier than download an extra 20MB (looking at you, Ethiopia). Telegram will try to download less data for users in such countries.

On the other hand, if you're likely to have a monster data allowance, Telegram will try to save you from tapping the "download" arrow too often.

These new default limits for automatic downloads are not set in stone. Starting today, we can change them remotely, based on your feedback and the cost of data in each country.

To Log Out or Not to Log Out?

New users often bring their logout habits from other apps and don't realize that this is rarely necessary on Telegram. To help them find their way around the app, the logout menu now shows several **alternative options** to logging out:



Logout Alternatives

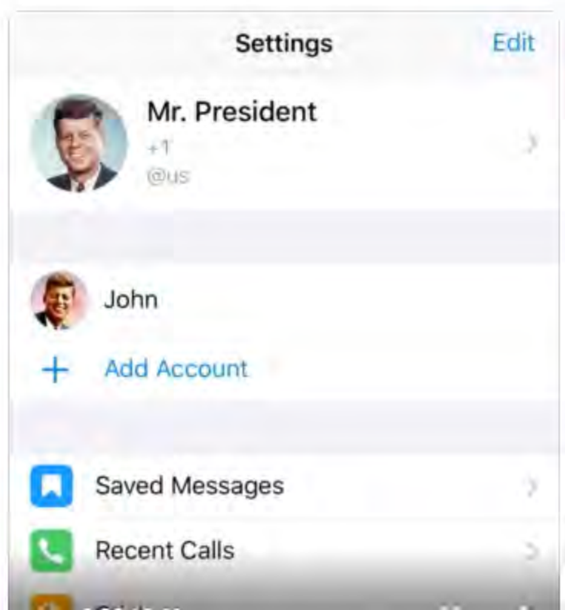
Multiple Account Support

Some of us have several phone numbers and multiple Telegram accounts: one for work, another strictly personal and a third one provided by benevolent aliens along with instructions to never use it except in a planetary emergency.

You can **add** all these accounts to your app and easily **switch** between up to **3 phone numbers** without logging out.

Add Account in Settings > Edit

If you've added several accounts, you will receive **push notifications** for all of them. Notifications will include information on which account they were sent to. You can also tap and hold on an account in Settings for a sneak peek of its chats list:



The multiple accounts feature was born in Telegram for Android and is now also available on iOS. We hope your alter-egos will be pleased.

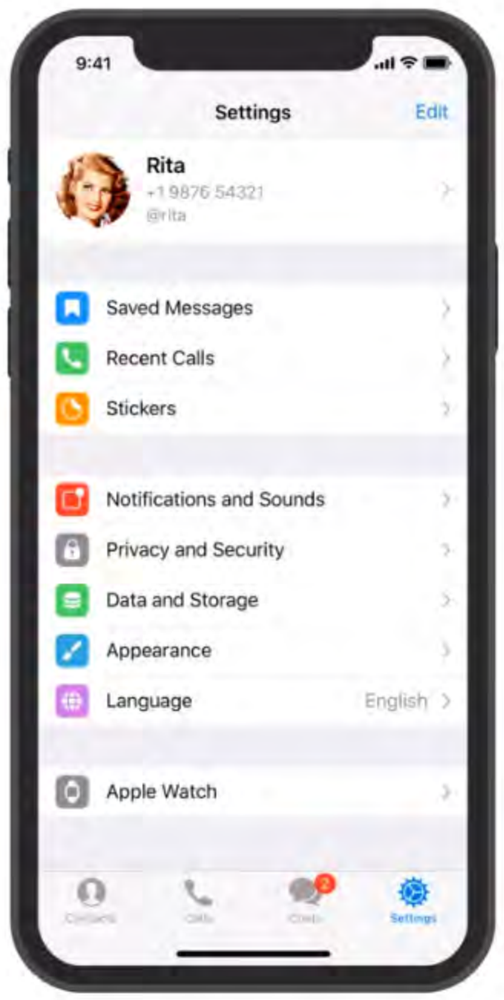
February 26, 2019,
The Telegram Team

Forward Tweet

Chat Backgrounds 2.0: Make Your Own

No chat can be dull if you have a really cool **chat background**. Today's update will ensure you get one. You can now search the web for wallpapers, add effects and then share your backgrounds with friends via links.

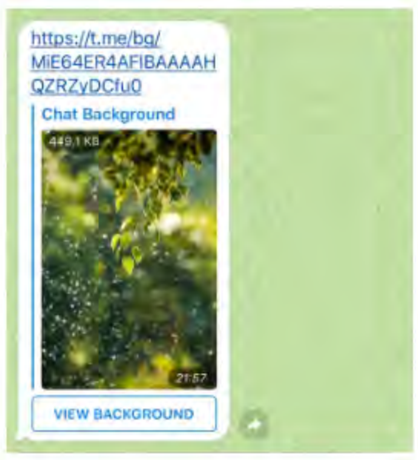
Telegram backgrounds now support **motion** and **blur** effects. You can also set **any color** as your background, apply a **pattern** and tweak its **intensity**.



Just like before, you can set **any photo** from your gallery as your chat background. On Android, you can add some extra effects in the built-in photo editor.

Share links

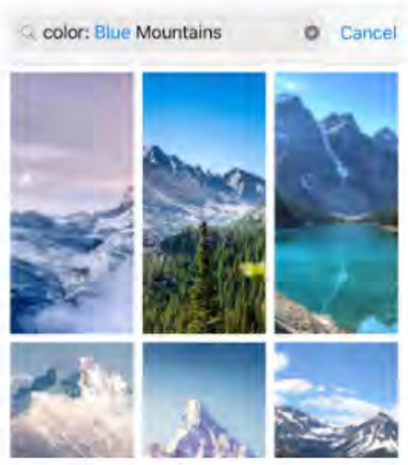
Having used this new arsenal to create the perfect chat background, you can easily set it to Telegram on all your other devices. What's more, you can infect the rest of the world with your genius by **sharing your background via a link**, just like this one: https://t.me/bg/l7jg-vpxmEYBAAAA1e0rNKySlkk



Search backgrounds

If you don't feel very creative and would like to simply set something nice real quick, we've **added new backgrounds** to the official selection.

We've also included a **background search** option. Find your dream wallpaper based on keywords or color and then set it in just a few taps.



We're also announcing the **Instant View 2.0 Template Competition**: two months, **$300,000+** in prizes, **$100** per template. See Instant View Contest 2.0 for details.

January 31, 2019
The Telegram Team

Forward Tweet

Older →

MTProto Mobile Protocol

> Please feel free to check out our FAQ for the Technically Inclined.
> Client developers are required to comply with the Security Guidelines.

Related articles

Mobile Protocol: Detailed Description
Creating an Authorization Key
Creating an Authorization Key: Example
Mobile Protocol: Service Messages
Mobile Protocol: Service Messages about Messages
Binary Data Serialization
TL Language

MTProto TL-schema

End-to-end encryption, Secret Chats
End-to-end TL-schema
Security Guidelines for Client Software Developers

This page deals with the basic layer of MTProto encryption used for Cloud chats (server-client encryption). See also:

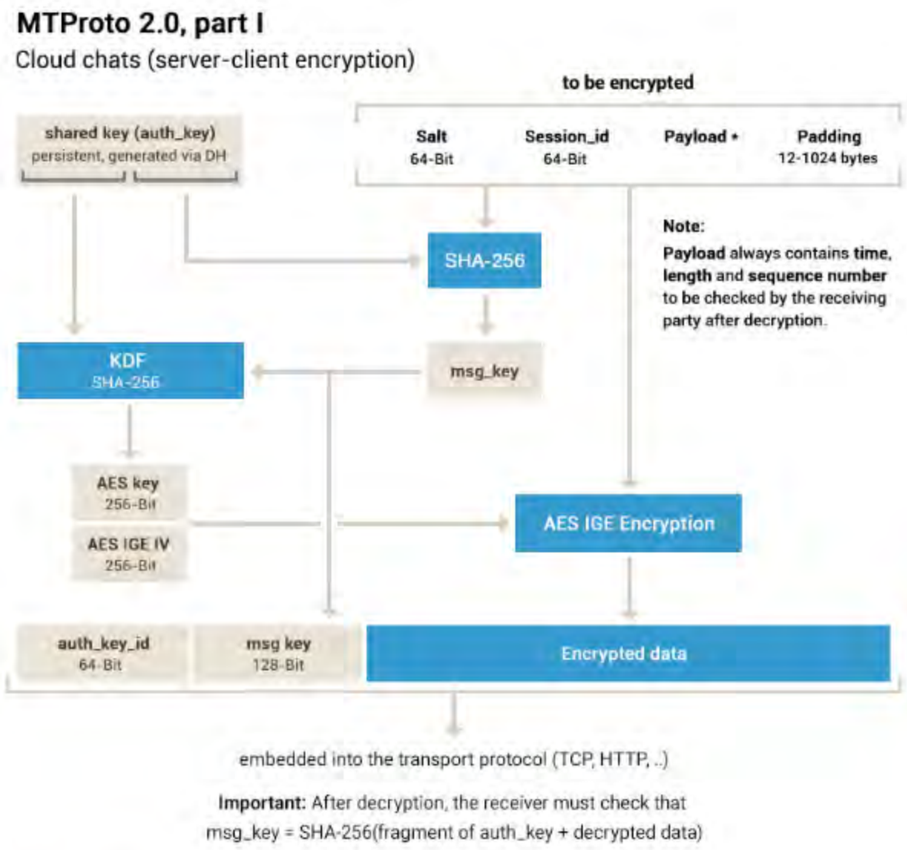
- Secret Chats, end-to-end-encryption
- End-to-end encrypted Voice Calls

General Description

The protocol is designed for access to a server API from applications running on mobile devices. It must be emphasized that a web browser is not such an application.

The protocol is subdivided into three virtually independent components:

- High-level component (API query language): defines the method whereby API queries and responses are converted to binary *messages*.
- Cryptographic (authorization) layer: defines the method by which messages are encrypted prior to being transmitted through the transport protocol.
- Transport component: defines the method for the client and the server to transmit messages over some other existing network protocol (such as HTTP, HTTPS, WS (plain websockets), WSS (websockets over HTTPS), TCP, UDP).



> As of version 4.6, major Telegram clients are using **MTProto 2.0**, described in this article.
> MTProto v1.0 (described here for reference) is deprecated and is currently being phased out.

Brief Component Summary

High-Level Component (RPC Query Language/API)

From the standpoint of the high-level component, the client and the server exchange *messages* inside a *session*. The session is attached to the client device (the application, to be more exact) rather than a specific websocket/http/https/tcp connection. In addition, each session is attached to a *user key ID* by which authorization is actually accomplished.

Several connections to a server may be open; messages may be sent in either direction through any of the connections (a response to a query is not necessarily returned through the same connection that carried the original query, although most often, that is the case; however, in no case can a message be returned through a connection belonging to a different session). When the UDP protocol is used, a response might be returned by a different IP address than the one to which the query had been sent.

There are several types of messages:

- RPC calls (client to server): calls to API methods
- RPC responses (server to client): results of RPC calls
- Message received acknowledgment (or rather, notification of status of a set of messages)
- Message status query
- *Multipart message* or *container* (a container that holds several messages; needed to send several RPC calls at once over an HTTP connection, for example; also, a container may support gzip).

From the standpoint of lower level protocols, a message is a binary data stream aligned along a 4 or 16-byte boundary. The first several fields in the message are fixed and are used by the cryptographic/authorization system.

Each message, either individual or inside a container, consists of a *message identifier* (64 bits, see below), a *message sequence number within a session* (32 bits), the *length* (of the message body in bytes; 32 bits), and a *body* (any size which is a multiple of 4 bytes). In addition, when a container or a single message is sent, an *internal header* is added at the top (see below), then the entire message is encrypted, and an *external header* is placed at the top of the message (a 64-bit *key identifier* and a 128-bit *message key*).

A *message body* normally consists of a 32-bit *message type* followed by type-dependent *parameters*. In particular, each RPC function has a corresponding message type. For more detail, see Binary Data Serialization, Mobile Protocol: Service Messages.

All numbers are written as little endian. However, very large numbers (2048-bit) used in RSA and DH are written in the big endian format because that's how the OpenSSL library does it.

Authorization and Encryption

Prior to a message (or a multipart message) being transmitted over a network using a transport protocol, it is encrypted in a certain way, and an *external header* is added at the top of the message which is: a 64-bit *key identifier* (that uniquely identifies an *authorization key* for the server as well as the *user*) and a 128-bit *message key*. A user key together with the message key defines an actual 256-bit key which is what encrypts the message using AES-256 encryption. Note that the initial part of the message to be encrypted contains variable data (session, message ID, sequence number, server salt) that obviously influences the message key (and thus the AES key and iv). The message key is defined as the 128 middle bits of the SHA256 of the message body (including session, message ID, etc.), including the padding bytes, prepended by 32 bytes taken from the authorization key. Multipart messages are encrypted as a single message.

> For a technical specification, see Mobile Protocol: Detailed Description

The first thing a client application must do is create an authorization key which is normally generated when it is first run and almost never changes.

The protocol's principal drawback is that an intruder passively intercepting messages and then somehow appropriating the authorization key (for example, by stealing a device) will be able to decrypt all the intercepted messages *post factum*. This probably is not too much of a problem (by stealing a device, one could also gain access to all the information cached on the device without decrypting anything); however, the following steps could be taken to overcome this weakness:

- *Session keys* generated using the Diffie-Hellman protocol and used in conjunction with the authorization and the message keys to select AES parameters. To create these, the first thing a client must do after creating a new session is send a special RPC query to the server ("generate session key") to which the server will respond, whereupon all subsequent messages within the session are encrypted using the session key as well.
- Protecting the key stored on the client device with a (text) password; this password is never stored in memory and is entered by a user when starting the application or more frequently (depending on application settings).
- Data stored (cached) on the user device can also be protected by encryption using an authorization key which, in turn, is to be password-protected. Then, a password will be required to gain access even to that data.

Time Synchronization

If client time diverges widely from server time, a server may start ignoring client messages, or vice versa, because of an invalid message identifier (which is closely related to creation time). Under these circumstances, the server will send the client a special message containing the correct time and a certain 128-bit salt (either explicitly provided by the client in a special RPC synchronization request or equal to the key of the latest message received from the client during the current session). This message could be the first one in a container that includes other messages (if the time discrepancy is significant but does not as yet result in the client's messages being ignored).

Having received such a message or a container holding it, the client first performs a time synchronization (in effect, simply storing the difference between the server's time and its own to be able to compute the "correct" time in the future) and then verifies that the message identifiers for correctness.

Where a correction has been neglected, the client will have to generate a new session to assure the monotonicity of message identifiers.

MTProto transport

Before being sent using the selected transport protocol, the payload has to be wrapped in a secondary protocol header, defined by the appropriate MTProto transport protocol.

- Abridged
- Intermediate
- Padded intermediate
- Full

The server recognizes these different protocols (and distinguishes them from HTTP, too) by the header.
Additionally, the following transport features can be used:

- Quick ack
- Transport errors
- Transport obfuscation

Example implementations for these protocols can be seen in tdlib and MadelineProto.

Transport

Enables the delivery of encrypted containers together with the external header (hereinafter, *Payload*) from client to server and back.
Multiple transport protocols are defined:

- TCP
- Websocket
- Websocket over HTTPS
- HTTP
- HTTPS
- UDP

(We shall examine only the first five types.)

Recap

To recap, using the ISO/OSI stack as comparison:

- Layer 7 (Application): High-level RPC API
- Layer 6 (Presentation): Type Language
- Layer 5 (Session): MTProto session
- Layer 4 (Transport):
  - 4.3: MTProto transport protocol
  - 4.2: MTProto obfuscation (optional)
  - 4.1: Transport protocol
- Layer 3 (Network): IP
- Layer 2 (Data link): MAC/LLC
- Layer 1 (Physical): IEEE 802.3, IEEE 802.11, etc...

Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About
FAQ
Blog
Jobs

Mobile Apps
iPhone/iPad
Android
Windows Phone

Desktop Apps
PC/Mac/Linux
macOS
Web-browser

Platform
API
Translations
Instant View

Mobile Protocol: Detailed Description

> As of version 4.6, major Telegram clients are using **MTProto 2.0**.
> MTProto v.1.0 is deprecated and is currently being phased out.

This article describes the basic layer of the MTProto protocol version 2.0 (Cloud chats, server-client encryption). The principal differences from version 1.0 (described here for reference) are as follows:

- SHA-256 is used instead of SHA-1;
- Padding bytes are involved in the computation of **msg_key**;
- **msg_key** depends not only on the message to be encrypted, but on a portion of **auth_key** as well;
- 12..1024 padding bytes are used instead of 0..15 padding bytes in v.1.0.

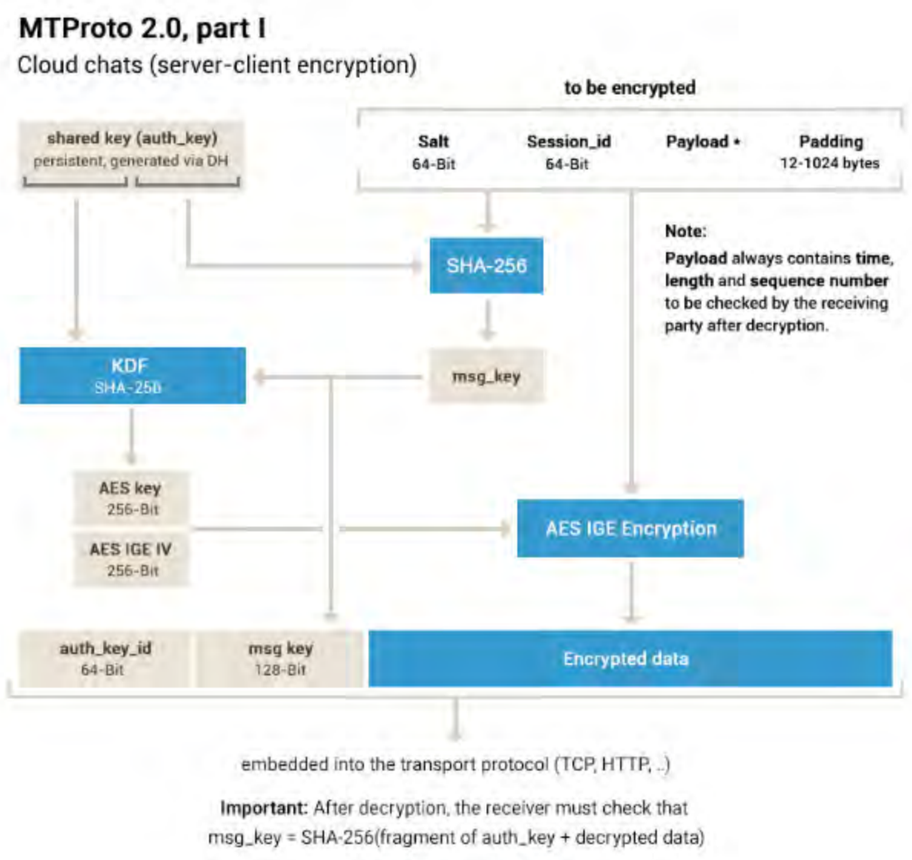
See also: MTProto 2.0: Secret Chats, end-to-end encryption

Protocol description

Before a message (or a multipart message) is transmitted over a network using a transport protocol, it is encrypted in a certain way, and an external header is added at the top of the message that consists of a 64-bit key identifier **auth_key_id** (that uniquely identifies an authorization key for the server as well as the user) and a 128-bit message key **msg_key**.

The authorization key **auth_key** combined with the message key **msg_key** define an actual 256-bit key **aes_key** and a 256-bit initialization vector **aes_iv**, which are used to encrypt the message using AES-256 encryption in infinite garble extension (IGE) mode. Note that the initial part of the message to be encrypted contains variable data (session, message ID, sequence number, server salt) that obviously influences the message key (and thus the AES key and iv). In **MTProto 2.0**, the message key is defined as the 128 middle bits of the SHA-256 of the message body (including session, message ID, padding, etc.) prepended by 32 bytes taken from the authorization key. In the older **MTProto 1.0**, the message key was computed as the lower 128 bits of SHA-1 of the message body, excluding the padding bytes.

Multipart messages are encrypted as a single message.



> Got questions about this setup? — Check out the Advanced FAQ!

Note 1

Each plaintext message to be encrypted in MTProto always contains the following data to be checked upon decryption in order to make the system robust against known problems with the components:

- server salt (64-Bit)
- session id
- message sequence number
- message length
- time

Note 2

Telegram's **End-to-end** encrypted Secret Chats are using an additional layer of encryption on top of the described above. See Secret Chats, End-to-End encryption for details.

Terminology

Authorization Key (auth_key)

A 2048-bit key shared by the client device and the server, created upon user registration directly on the client device by exchanging Diffie-Hellman keys, and never transmitted over a network. Each authorization key is user-specific. There is nothing that prevents a user from having several keys (that correspond to "permanent sessions" on different devices), and some of these may be locked forever in the event the device is lost. See also Creating an Authorization Key.

Server Key

A 2048-bit RSA key used by the server digitally to sign its own messages while registration is underway and the authorization key is being generated. The application has a built-in public server key which can be used to verify a signature but cannot be used to sign messages. A private server key is stored on the server and changed very infrequently.

Key Identifier (auth_key_id)

The 64 lower-order bits of the SHA1 hash of the authorization key are used to indicate which particular key was used to encrypt a message. Keys must be uniquely defined by the 64 lower-order bits of their SHA1, and in the event of a collision, an authorization key is regenerated. A zero key identifier means that encryption is not used which is permissible for a limited set of message types used during registration to generate an authorization key in a Diffie-Hellman exchange. **For MTProto 2.0, SHA1 is still used here, because auth_key_id should identify the authorization key used independently of the protocol version.**

Session

A (random) 64-bit number generated by the client to distinguish between individual sessions (for example, between different instances of the application, created with the same authorization key). The session in conjunction with the key identifier corresponds to an application instance. The server can maintain session state. *Under no circumstances*

can a message meant for one session be sent into a different session. The server may unilaterally forget any client sessions; clients should treat this event as permanent.

Server Salt

A (random) 64-bit number periodically (say, every 24 hours) changed (separately for each session) at the request of the server. All subsequent messages must contain the new salt (although, messages with the old salt are still accepted for a further 300 seconds). Required to protect against replay attacks and certain tricks associated with adjusting the client clock to a moment in the distant future.

Message Identifier (msg_id)

A (time-dependent) 64-bit number used uniquely to identify a message within a session. Client message identifiers are divisible by 4, server message identifiers modulo 4 yield 1 if the message is a response to a client message, and 3 otherwise. Client message identifiers must increase monotonically (within a single session), the same as server message identifiers, and must approximately equal unixtime*2^32. This way, a message identifier points to the approximate moment in time the message was created. A message is rejected over 300 seconds after it is created or 30 seconds before it is created (this is needed to protect from replay attacks). In this situation, it must be re-sent with a different identifier (or placed in a container with a higher identifier). The identifier of a message container must be strictly greater than those of its nested messages.

Important: to counter replay-attacks the lower 32 bits of **msg_id** passed by the client must not be empty and must present a fractional part of the time point when the message was created.

Content-related Message

A message requiring an explicit acknowledgment. These include all the user and many service messages, virtually all with the exception of containers and acknowledgments.

Message Sequence Number (msg_seqno)

A 32-bit number equal to twice the number of "content-related" messages (those requiring acknowledgment, and in particular those that are not containers) created by the sender prior to this message and subsequently incremented by one if the current message is a content-related message. A container is always generated after its entire contents; therefore, its sequence number is greater than or equal to the sequence numbers of the messages contained in it.

Message Key (msg_key)

In **MTProto 2.0**, the middle 128 bits of the SHA-256 hash of the message to be encrypted (including the internal header and the *padding bytes* for MTProto 2.0), prepended by a 32-byte fragment of the authorization key.

In **MTProto 1.0**, message key was defined differently, as the lower 128 bits of the SHA-1 hash of the message to be encrypted, with padding bytes excluded from the computation of the hash. Authorization key was not involved in this computation.

Internal (cryptographic) Header

A header (16 bytes) added before a message or a container before it is all encrypted together. Consists of the server salt (64 bits) and the session (64 bits).

External (cryptographic) Header

A header (24 bytes) added before an encrypted message or a container. Consists of the key identifier **auth_key_id** (64 bits) and the message key **msg_key** (128 bits).

Payload

External header + encrypted message or container.

Defining AES Key and Initialization Vector

The 2048-bit authorization key (auth_key) and the 128-bit message key (msg_key) are used to compute a 256-bit AES key (aes_key) and a 256-bit initialization vector (aes_iv) which are subsequently used to encrypt the part of the message to be encrypted (i. e. everything with the exception of the external header that is added later) with AES-256 in infinite garble extension (IGE) mode.

For MTProto 2.0, the algorithm for computing aes_key and aes_iv from auth_key and msg_key is as follows.

- msg_key_large = SHA256 (substr (auth_key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (auth_key, x, 36));
- sha256_b = SHA256 (substr (auth_key, 40+x, 36) + msg_key);
- aes_key = substr (sha256_a, 0, 8) + substr (sha256_b, 8, 16) + substr (sha256_a, 24, 8);
- aes_iv = substr (sha256_b, 0, 8) + substr (sha256_a, 8, 16) + substr (sha256_b, 24, 8);

where x = 0 for messages from client to server and x = 8 for those from server to client.

For the obsolete MTProto 1.0, msg_key, aes_key, and aes_iv were computed differently (see this document for reference).

The lower-order 1024 bits of auth_key are not involved in the computation. They may (together with the remaining bits or separately) be used on the client device to encrypt the local copy of the data received from the server. The 512 lower-order bits of auth_key are not stored on the server; therefore, if the client device uses them to encrypt local data and the user loses the key or the password, data decryption of local data is impossible (even if data from the server could be obtained).

In MTProto 1.0, when AES was used to encrypt a block of data of a length not divisible by 16 bytes, the data was padded with 0 to 15 random padding bytes **random_padding** to a length divisible by 16 bytes prior to encryption. **In MTProto 2.0, this padding is taken into account when computing `msg_key`. Note that MTProto 2.0 requires from 12 to 1024 bytes of padding, still subject to the condition that the resulting message length be divisible by 16 bytes.**

Using MTProto 2.0 instead of MTProto 1.0

A client may either use only MTProto 2.0 or only MTProto 1.0 in the same TCP connection. The server detects the protocol used by the first message received from the client, and then uses the same encryption for its messages, and expects the client to use the same encryption henceforth. We recommend using MTProto 2.0; MTProto 1.0 is deprecated and supported for backward compatibility only.

Important Checks

When an encrypted message is received, it *must* be checked that **msg_key** is *in fact* equal to the 128 middle bits of the SHA-256 of the decrypted data with a 32-byte fragment of **auth_key** prepended to it, and that msg_id has even parity for messages from client to server, and odd parity for messages from server to client.

In addition, the identifiers (msg_id) of the last N messages received from the other side must be stored, and if a message comes in with msg_id lower than all or equal to any of the stored values, the message is to be ignored. Otherwise, the new message msg_id is added to the set, and, if the number of stored msg_id values is greater than N, the oldest (i. e. the lowest) is forgotten.

On top of this, msg_id values that belong over 30 seconds in the future or over 300 seconds in the past are to be ignored. This is especially important for the server. The client would also find this useful (to protect from a replay attack), but only if it is certain of its time (for example, if its time has been synchronized with that of the server).

Certain client-to-server service messages containing data sent by the client to the server (for example, msg_id of a recent client query) may, nonetheless, be processed on the client even if the time appears to be "incorrect". This is especially true of messages to change server_salt and notifications of invalid client time. See Mobile Protocol: Service Messages.

Storing an Authorization Key on a Client Device

It may be suggested to users concerned with security that they password protect the authorization key in approximately the same way as in ssh. This can be accomplished by prepending the value of cryptographic hash

function, such as SHA-256, of the key to the front of the key, following which the entire string is encrypted using AES in CBC mode and a key equal to the user's (text) password. When the user inputs the password, the stored protected password is decrypted and verified by checking the SHA-256 value. From the user's standpoint, this is practically the same as using an application or a website password.

Unencrypted Messages

Special plain-text messages may be used to create an authorization key as well as to perform a time synchronization. They begin with auth_key_id = 0 (64 bits) which means that there is no auth_key. This is followed directly by the message body in serialized format without internal or external headers. A message identifier (64 bits) and body length in bytes (32 bytes) are added before the message body.

Only a very limited number of messages of special types can be transmitted as plain text.

Schematic Presentation of Messages

Encrypted Message

auth_key_id int64	msg_key int128	encrypted_data bytes

Encrypted Message: *encrypted_data*

Contains the cypher text for the following data:

salt int64	session_id int64	message_id int64	seq_no int32	message_data_length int32	message_data bytes	padding12..1024 bytes

Unencrypted Message

auth_key_id = `0` int64	message_id int64	message_data_length int32	message_data bytes

MTProto 2.0 uses 12..1024 padding bytes, instead of the 0..15 used in MTProto 1.0

Creating an Authorization Key

An authorization key is normally created once for every user during the application installation process immediately prior to registration. Registration itself, in actuality, occurs after the authorization key is created. However, a user may be prompted to complete the registration form while the authorization key is being generated in the background. Intervals between user key strokes may be used as a source of entropy in the generation of high-quality random numbers required for the creation of an authorization key.

See Creating an Authorization Key.

During the creation of the authorization key, the client obtains its server salt (to be used with the new key for all communication in the near future). The client then creates an encrypted session using the newly generated key, and subsequent communication occurs within that session (including the transmission of the user's registration information and phone number validation) unless the client creates a new session. The client is free to create new or additional sessions at any time by choosing a new random session_id.

Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About
FAQ
Blog
Jobs

Mobile Apps
iPhone/iPad
Android
Windows Phone

Desktop Apps
PC/Mac/Linux
macOS
Web-browser

Platform
API
Translations
Instant View

Mobile Protocol: Detailed Description

> As of version 4.6, major Telegram clients are using **MTProto 2.0**.
> MTProto v.1.0 is deprecated and is currently being phased out.

This article describes the basic layer of the MTProto protocol version 2.0 (Cloud chats, server-client encryption). The principal differences from version 1.0 (described here for reference) are as follows:

- SHA-256 is used instead of SHA-1;
- Padding bytes are involved in the computation of **msg_key**;
- **msg_key** depends not only on the message to be encrypted, but on a portion of **auth_key** as well;
- 12..1024 padding bytes are used instead of 0..15 padding bytes in v.1.0.

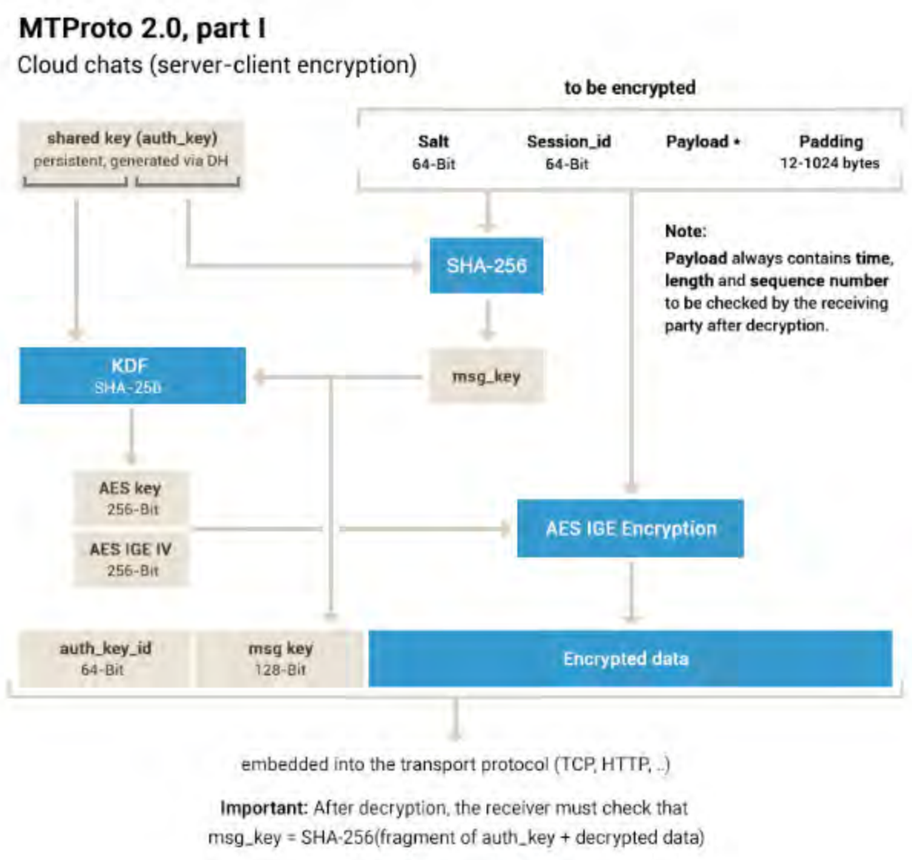
See also: MTProto 2.0: Secret Chats, end-to-end encryption

Protocol description

Before a message (or a multipart message) is transmitted over a network using a transport protocol, it is encrypted in a certain way, and an external header is added at the top of the message that consists of a 64-bit key identifier **auth_key_id** (that uniquely identifies an authorization key for the server as well as the user) and a 128-bit message key **msg_key**.

The authorization key **auth_key** combined with the message key **msg_key** define an actual 256-bit key **aes_key** and a 256-bit initialization vector **aes_iv**, which are used to encrypt the message using AES-256 encryption in infinite garble extension (IGE) mode. Note that the initial part of the message to be encrypted contains variable data (session, message ID, sequence number, server salt) that obviously influences the message key (and thus the AES key and iv). In **MTProto 2.0**, the message key is defined as the 128 middle bits of the SHA-256 of the message body (including session, message ID, padding, etc.) prepended by 32 bytes taken from the authorization key. In the older **MTProto 1.0**, the message key was computed as the lower 128 bits of SHA-1 of the message body, excluding the padding bytes.

Multipart messages are encrypted as a single message.



> Got questions about this setup? — Check out the Advanced FAQ!

Note 1

Each plaintext message to be encrypted in MTProto always contains the following data to be checked upon decryption in order to make the system robust against known problems with the components:

- server salt (64-Bit)
- session id
- message sequence number
- message length
- time

Note 2

Telegram's **End-to-end** encrypted Secret Chats are using an additional layer of encryption on top of the described above. See Secret Chats, End-to-End encryption for details.

Terminology

Authorization Key (auth_key)

A 2048-bit key shared by the client device and the server, created upon user registration directly on the client device by exchanging Diffie-Hellman keys, and never transmitted over a network. Each authorization key is user-specific. There is nothing that prevents a user from having several keys (that correspond to "permanent sessions" on different devices), and some of these may be locked forever in the event the device is lost. See also Creating an Authorization Key.

Server Key

A 2048-bit RSA key used by the server digitally to sign its own messages while registration is underway and the authorization key is being generated. The application has a built-in public server key which can be used to verify a signature but cannot be used to sign messages. A private server key is stored on the server and changed very infrequently.

Key Identifier (auth_key_id)

The 64 lower-order bits of the SHA1 hash of the authorization key are used to indicate which particular key was used to encrypt a message. Keys must be uniquely defined by the 64 lower-order bits of their SHA1, and in the event of a collision, an authorization key is regenerated. A zero key identifier means that encryption is not used which is permissible for a limited set of message types used during registration to generate an authorization key in a Diffie-Hellman exchange. **For MTProto 2.0, SHA1 is still used here, because auth_key_id should identify the authorization key used independently of the protocol version.**

Session

A (random) 64-bit number generated by the client to distinguish between individual sessions (for example, between different instances of the application, created with the same authorization key). The session in conjunction with the key identifier corresponds to an application instance. The server can maintain session state. *Under no circumstances*

can a message meant for one session be sent into a different session. The server may unilaterally forget any client sessions; clients should treat this event as a network error.

Server Salt

A (random) 64-bit number periodically (say, every 24 hours) changed (separately for each session) at the request of the server. All subsequent messages must contain the new salt (although, messages with the old salt are still accepted for a further 300 seconds). Required to protect against replay attacks and certain tricks associated with adjusting the client clock to a moment in the distant future.

Message Identifier (msg_id)

A (time-dependent) 64-bit number used uniquely to identify a message within a session. Client message identifiers are divisible by 4, server message identifiers modulo 4 yield 1 if the message is a response to a client message, and 3 otherwise. Client message identifiers must increase monotonically (within a single session), the same as server message identifiers, and must approximately equal unixtime*2^32. This way, a message identifier points to the approximate moment in time the message was created. A message is rejected over 300 seconds after it is created or 30 seconds before it is created (this is needed to protect from replay attacks). In this situation, it must be re-sent with a different identifier (or placed in a container with a higher identifier). The identifier of a message container must be strictly greater than those of its nested messages.

Important: to counter replay-attacks the lower 32 bits of **msg_id** passed by the client must not be empty and must present a fractional part of the time point when the message was created.

Content-related Message

A message requiring an explicit acknowledgment. These include all the user and many service messages, virtually all with the exception of containers and acknowledgments.

Message Sequence Number (msg_seqno)

A 32-bit number equal to twice the number of "content-related" messages (those requiring acknowledgment, and in particular those that are not containers) created by the sender prior to this message and subsequently incremented by one if the current message is a content-related message. A container is always generated after its entire contents; therefore, its sequence number is greater than or equal to the sequence numbers of the messages contained in it.

Message Key (msg_key)

In **MTProto 2.0**, the middle 128 bits of the SHA-256 hash of the message to be encrypted (including the internal header and the *padding bytes* for MTProto 2.0), prepended by a 32-byte fragment of the authorization key.

In **MTProto 1.0**, message key was defined differently, as the lower 128 bits of the SHA-1 hash of the message to be encrypted, with padding bytes excluded from the computation of the hash. Authorization key was not involved in this computation.

Internal (cryptographic) Header

A header (16 bytes) added before a message or a container before it is all encrypted together. Consists of the server salt (64 bits) and the session (64 bits).

External (cryptographic) Header

A header (24 bytes) added before an encrypted message or a container. Consists of the key identifier **auth_key_id** (64 bits) and the message key **msg_key** (128 bits).

Payload

External header + encrypted message or container.

Defining AES Key and Initialization Vector

The 2048-bit authorization key (auth_key) and the 128-bit message key (msg_key) are used to compute a 256-bit AES key (aes_key) and a 256-bit initialization vector (aes_iv) which are subsequently used to encrypt the part of the message to be encrypted (i. e. everything with the exception of the external header that is added later) with AES-256 in infinite garble extension (IGE) mode.

For MTProto 2.0, the algorithm for computing aes_key and aes_iv from auth_key and msg_key is as follows.

- msg_key_large = SHA256 (substr (auth_key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (auth_key, x, 36));
- sha256_b = SHA256 (substr (auth_key, 40+x, 36) + msg_key);
- aes_key = substr (sha256_a, 0, 8) + substr (sha256_b, 8, 16) + substr (sha256_a, 24, 8);
- aes_iv = substr (sha256_b, 0, 8) + substr (sha256_a, 8, 16) + substr (sha256_b, 24, 8);

where x = 0 for messages from client to server and x = 8 for those from server to client.

For the obsolete MTProto 1.0, msg_key, aes_key, and aes_iv were computed differently (see this document for reference).

The lower-order 1024 bits of auth_key are not involved in the computation. They may (together with the remaining bits or separately) be used on the client device to encrypt the local copy of the data received from the server. The 512 lower-order bits of auth_key are not stored on the server; therefore, if the client device uses them to encrypt local data and the user loses the key or the password, data decryption of local data is impossible (even if data from the server could be obtained).

In MTProto 1.0, when AES was used to encrypt a block of data of a length not divisible by 16 bytes, the data was padded with 0 to 15 random padding bytes **random_padding** to a length divisible by 16 bytes prior to encryption. **In MTProto 2.0, this padding is taken into account when computing `msg_key`. Note that MTProto 2.0 requires from 12 to 1024 bytes of padding, still subject to the condition that the resulting message length be divisible by 16 bytes.**

Using MTProto 2.0 instead of MTProto 1.0

A client may either use only MTProto 2.0 or only MTProto 1.0 in the same TCP connection. The server detects the protocol used by the first message received from the client, and then uses the same encryption for its messages, and expects the client to use the same encryption henceforth. We recommend using MTProto 2.0; MTProto 1.0 is deprecated and supported for backward compatibility only.

Important Checks

When an encrypted message is received, it *must* be checked that **msg_key** is *in fact* equal to the 128 middle bits of the SHA-256 of the decrypted data with a 32-byte fragment of **auth_key** prepended to it, and that msg_id has even parity for messages from client to server, and odd parity for messages from server to client.

In addition, the identifiers (msg_id) of the last N messages received from the other side must be stored, and if a message comes in with msg_id lower than all or equal to any of the stored values, the message is to be ignored. Otherwise, the new message msg_id is added to the set, and, if the number of stored msg_id values is greater than N, the oldest (i. e. the lowest) is forgotten.

On top of this, msg_id values that belong over 30 seconds in the future or over 300 seconds in the past are to be ignored. This is especially important for the server. The client would also find this useful (to protect from a replay attack), but only if it is certain of its time (for example, if its time has been synchronized with that of the server).

Certain client-to-server service messages containing data sent by the client to the server (for example, msg_id of a recent client query) may, nonetheless, be processed on the client even if the time appears to be "incorrect". This is especially true of messages to change server_salt and notifications of invalid client time. See Mobile Protocol: Service Messages.

Storing an Authorization Key on a Client Device

It may be suggested to users concerned with security that they password protect the authorization key in approximately the same way as in ssh. This can be accomplished by prepending the value of cryptographic hash

function, such as SHA-256, of the key to the front of the key, following which the entire string is encrypted using AES in CBC mode and a key equal to the user's (text) password. When the user inputs the password, the stored protected password is decrypted and verified by checking the SHA-256 value. From the user's standpoint, this is practically the same as using an application or a website password.

Unencrypted Messages

Special plain-text messages may be used to create an authorization key as well as to perform a time synchronization. They begin with auth_key_id = 0 (64 bits) which means that there is no auth_key. This is followed directly by the message body in serialized format without internal or external headers. A message identifier (64 bits) and body length in bytes (32 bytes) are added before the message body.

Only a very limited number of messages of special types can be transmitted as plain text.

Schematic Presentation of Messages

Encrypted Message

auth_key_id int64	msg_key int128	encrypted_data bytes

Encrypted Message: *encrypted_data*

Contains the cypher text for the following data:

salt int64	session_id int64	message_id int64	seq_no int32	message_data_length int32	message_data bytes	padding12..1024 bytes

Unencrypted Message

auth_key_id = `0` int64	message_id int64	message_data_length int32	message_data bytes

MTProto 2.0 uses 12..1024 padding bytes, instead of the 0..15 used in MTProto 1.0

Creating an Authorization Key

An authorization key is normally created once for every user during the application installation process immediately prior to registration. Registration itself, in actuality, occurs after the authorization key is created. However, a user may be prompted to complete the registration form while the authorization key is being generated in the background. Intervals between user key strokes may be used as a source of entropy in the generation of high-quality random numbers required for the creation of an authorization key.

See Creating an Authorization Key.

During the creation of the authorization key, the client obtains its server salt (to be used with the new key for all communication in the near future). The client then creates an encrypted session using the newly generated key, and subsequent communication occurs within that session (including the transmission of the user's registration information and phone number validation) unless the client creates a new session. The client is free to create new or additional sessions at any time by choosing a new random session_id.

Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About
FAQ
Blog
Jobs

Mobile Apps
iPhone/iPad
Android
Windows Phone

Desktop Apps
PC/Mac/Linux
macOS
Web-browser

Platform
API
Translations
Instant View

Creating an Authorization Key

The query format is described using Binary Data Serialization and the TL Language. All large numbers are transmitted as strings containing the required sequence of bytes in big endian order. Hash functions, such as SHA1, return strings (of 20 bytes) which can also be interpreted as big endian numbers. Small numbers (`int` , `long` , `int128` , `int256`) are normally little endian; however, if they are part of SHA1, the bytes are not rearranged. This way, if `long x` is the 64 lower-order bits of SHA1 of string `s` , then the *final* 8 bytes of 20-byte string `SHA1(s)` are taken and interpreted as a 64-bit integer.

Prior to sending off unencrypted messages (required in this instance to generate an authorization key), the client must undergo (p,q) authorization as follows.

DH exchange initiation

1) Client sends query to server

```
req_pq_multi#be7e8ef1 nonce:int128 = ResPQ;
```

or (deprecated)

```
req_pq#60469778 nonce:int128 = ResPQ;
```

The value of *nonce* is selected randomly by the client (random number) and identifies the client within this communication. Following this step, it is known to all.

2) Server sends response of the form

```
resPQ#05162463 nonce:int128 server_nonce:int128 pq:string server_public_key_fingerprints:Vector long = ResPQ;
```

Here, string pq is a representation of a natural number (in binary big endian format). This number is the product of two different odd prime numbers. Normally, pq is less than or equal to 2^63-1. The value of *server_nonce* is selected randomly by the server; following this step, it is known to all.

`server_public_key_fingerprints` is a list of public RSA key fingerprints (64 lower-order bits of SHA1 (server_public_key); the public key is represented as a bare type `rsa_public_key n:string e:string = RSAPublicKey` , where, as usual, n and e are numbers in big endian format serialized as strings of bytes, following which SHA1 is computed) received by the server. Because of compatibility issues with older clients, only one public key fingerprint is returned as a result to deprecated `req_pq` query; an answer to `req_pq_multi` may contain more than one fingerprint.

All subsequent messages contain the pair (nonce, server_nonce) both in the plain-text, and the encrypted portions which makes it possible to identify a "temporary session" — one run of the key generation protocol described on this page that uses the same (nonce, server_nonce) pair. An intruder could not create a parallel session with the server with the same parameters and reuse parts of server- or client-encrypted messages for its own purposes in such a parallel session, because a different server_nonce would be selected by the server for any new "temporary session".

Proof of work

3) Client decomposes pq into prime factors such that p < q.

This starts a round of Diffie-Hellman key exchanges.

Presenting proof of work; Server authentication

4) Client sends query to server

```
req_DH_params#d712e4be nonce:int128 server_nonce:int128 p:string q:string public_key_fingerprint:long encrypted
```

Here, encrypted_data is obtained as follows:

- new_nonce := another (good) random number generated by the client; after this query, it is known to both client and server;
- data := a serialization of

  ```
  p_q_inner_data#83c95aec pq:string p:string q:string nonce:int128 server_nonce:int128 new_nonce:int256 = P_Q
  ```

  or of

  ```
  p_q_inner_data_temp#3c6a84d4 pq:string p:string q:string nonce:int128 server_nonce:int128 new_nonce:int256
  ```

- data_with_hash := SHA1(data) + data + (any random bytes); such that the length equal 255 bytes;
- encrypted_data := RSA (data_with_hash, server_public_key); a 255-byte long number (big endian) is raised to the requisite power over the requisite modulus, and the result is stored as a 256-byte number.

Someone might intercept the query and replace it with their own, independently decomposing pq into factors instead of the client. The only field that it makes sense to modify is new_nonce which would be the one an intruder would have to re-generate (because an intruder cannot decrypt the encrypted data sent by the client). Since all subsequent messages are encrypted using new_nonce or contain new_nonce_hash, they will not be processed by the client (an intruder would not be able to make it look as though they had been generated by the server because they would not contain new_nonce). Therefore, this intercept will only result in the intruder's completing the authorization key generation protocol in place of the client and creating a new key (that has nothing to do with the client); however, the same effect could be achieved simply by creating a new key in one's own name.

An alternative form of inner data (`p_q_inner_data_temp`) is used to create temporary keys, that are only stored in the server RAM and are discarded after at most `expires_in` seconds. The server is free to discard its copy earlier. In all other respects the temporary key generation protocol is the same. After a temporary key is created, the client usually binds it to its principal authorisation key by means of the auth.bindTempAuthKey method, and uses it for all client-server communication until it expires; then a new temporary key is generated. Thus Perfect Forward Secrecy (PFS) in client-server communication is achieved. Read more about PFS »

5) Server responds in one of two ways:

```
server_DH_params_fail#79cb045d nonce:int128 server_nonce:int128 new_nonce_hash:int128 = Server_DH_Params;
server_DH_params_ok#d0e8075c nonce:int128 server_nonce:int128 encrypted_answer:string = Server_DH_Params;
```

Here, encrypted_answer is obtained as follows:

- new_nonce_hash := 128 lower-order bits of SHA1 (new_nonce);
- answer := serialization

  ```
  server_DH_inner_data#b5890dba nonce:int128 server_nonce:int128 g:int dh_prime:string g_a:string server_ti
  ```

- answer_with_hash := SHA1(answer) + answer + (0-15 random bytes); such that the length be divisible by 16;
- tmp_aes_key := SHA1(new_nonce + server_nonce) + substr (SHA1(server_nonce + new_nonce), 0, 12);
- tmp_aes_iv := substr (SHA1(server_nonce + new_nonce), 12, 8) + SHA1(new_nonce + new_nonce) + substr (new_nonce, 0, 4);
- encrypted_answer := AES256_ige_encrypt (answer_with_hash, tmp_aes_key, tmp_aes_iv); here, tmp_aes_key is a 256-bit key, and tmp_aes_iv is a 256-bit initialization vector. The same as in all the other instances that use AES encryption, the encrypted data is padded with random bytes to a length divisible by 16 immediately prior to encryption.

Following this step, new_nonce is still known to client and server only. The client is certain that it is the server that responded and that the response was generated specifically in response to client query req_DH_params, since the response data are encrypted using new_nonce.

Client is expected to check whether prime **dh_prime** is a safe 2048-bit prime (meaning that both **dh_prime** and **(p-1)/2** are prime, and that 2^2047 < p < 2^2048), and that **g** generates a cyclic subgroup of prime order **(p-1)/2**, i.e. is a quadratic residue **mod p**. Since **g** is always equal to 2, 3, 4, 5, 6 or 7, this is easily done using quadratic reciprocity law, yielding a simple condition on **p mod 4g** — namely, **p mod 8 = 7** for **g = 2**; **p mod 3 = 2** for **g = 3**; no extra condition for **g = 4**; **p mod 5 = 1 or 4** for **g = 5**; **p mod 24 = 19 or 23** for **g = 6**; and **p mod 7 = 3, 5 or 6** for **g = 7**. After **g** and **p** have been checked by the client, it makes sense to cache the result, so as not to repeat lengthy computations in future.

If the verification takes too long time (which is the case for older mobile devices), one might initially run only 15 Miller—Rabin iterations for verifying primeness of **p** and **(p - 1)/2** with error probability not exceeding one billionth, and do more iterations later in the background.

Another optimization is to embed into the client application code a small table with some known "good" couples **(g,p)** (or just known safe primes **p**, since the condition on **g** is easily verified during execution), checked during code generation phase, so as to avoid doing such verification during runtime altogether. Server changes these values rarely, thus one usually has to put the current value of server's **dh_prime** into such a table. For example, current value of **dh_prime** equals (in big-endian byte order)

```
C7 1C AE B9 C6 B1 C9 04 8E 6C 52 2F 70 F1 3F 73 98 0D 40 23 8E 3E 21 C1 49 34 D0 37 56 3D 93 0F 48 19 8A 0A A7
```

6) Client computes random 2048-bit number *b* (using a sufficient amount of entropy) and sends the server a message

```
set_client_DH_params#f5045f1f nonce:int128 server_nonce:int128 encrypted_data:string = Set_client_DH_params_ans
```

Here, encrypted_data is obtained thus:

- g_b := pow(g, b) mod dh_prime;
- data := serialization

```
client_DH_inner_data#6643b654 nonce:int128 server_nonce:int128 retry_id:long g_b:string = Client_DH_Inner
```

- data_with_hash := SHA1(data) + data + (0-15 random bytes); such that length be divisible by 16;
- encrypted_data := AES256_ige_encrypt (data_with_hash, tmp_aes_key, tmp_aes_iv);

The retry_id field is equal to zero at the time of the first attempt; otherwise, it is equal to auth_key_aux_hash from the previous failed attempt (see Item 9).

7) Thereafter, auth_key equals `pow(g, {ab}) mod dh_prime`; on the server, it is computed as `pow(g_b, a) mod dh_prime`, and on the client as `(g_a)^b mod dh_prime`.

8) auth_key_hash is computed := 64 lower-order bits of SHA1 (auth_key). The server checks whether there already is another key with the same auth_key_hash and responds in one of the following ways.

DH key exchange complete

9) Server responds in one of three ways:

```
dh_gen_ok#3bcbf734 nonce:int128 server_nonce:int128 new_nonce_hash1:int128 = Set_client_DH_params_answer;
dh_gen_retry#46dc1fb9 nonce:int128 server_nonce:int128 new_nonce_hash2:int128 = Set_client_DH_params_answer;
dh_gen_fail#a69dae02 nonce:int128 server_nonce:int128 new_nonce_hash3:int128 = Set_client_DH_params_answer;
```

- new_nonce_hash1, new_nonce_hash2, and new_nonce_hash3 are obtained as the 128 lower-order bits of SHA1 of the byte string derived from the new_nonce string by adding a single byte with the value of 1, 2, or 3, and followed by another 8 bytes with auth_key_aux_hash. Different values are required to prevent an intruder from changing server response dh_gen_ok into dh_gen_retry.
- auth_key_aux_hash is the 64 *higher-order* bits of SHA1(auth_key). It must not be confused with auth_key_hash.

In the other case, the client goes to Item 6) generating a new *b*.
In the first case, the client and the server have negotiated auth_key, following which they forget all other temporary data, and the client creates another encrypted session using auth_key. At the same time, server_salt is initially set to `substr(new_nonce, 0, 8) XOR substr(server_nonce, 0, 8)`. If required, the client stores the difference between server_time received in 5) and its local time, to be able always to have a good approximation of server time which is required to generate correct message identifiers.

IMPORTANT: Apart from the conditions on the Diffie-Hellman prime **dh_prime** and generator **g**, both sides are to check that **g**, **g_a** and **g_b** are greater than **1** and less than **dh_prime - 1**. We recommend checking that **g_a** and **g_b** are between **2^{2048-64}** and **dh_prime - 2^{2048-64}** as well.

Error Handling (Lost Queries and Responses)

If the client fails to receive any response to its query from the server within a certain time interval, it may simply re-send the query. If the server has already sent a response to this query (*exactly* the same request and not just similar: all the parameters during the repeat request must take on the same values) but it did not get to the client, the server will simply re-send the same response. The server remembers the response for up to 10 minutes after having received the query in 1). If the server has already forgotten the response or the requisite temporary data, the client will have to start from the beginning.

The server may consider that if the client has already sent in the next query using the data from the previous server response to the specific client, the response is known to have been received by the client and may be forgotten by the server.

Usage Example

An example of a complete list of queries required to generate an authorization key is shown on a separate page.

Telegram
Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About
FAQ
Blog
Jobs

Mobile Apps
iPhone/iPad
Android
Windows Phone

Desktop Apps
PC/Mac/Linux
macOS
Web-browser

Platform
API
Translations
Instant View

samples-auth_key

In the examples below, the transport headers are omitted:

> If a payload (packet) needs to be transmitted from server to client or from client to server, it is encapsulated as follows: 4 bytes are added at the front (to include the length, the sequence number, and CRC32; always divisible by 4) and 4 bytes with the packet sequence number for this TCP connection (the first packet sent is numbered 0, the next one 1, etc.), and 4 CRC32 bytes at the end (length, sequence number, and payload together).
>
> There is an abridged version of the same protocol: if the client sends `0xef` as the first byte (**important**: only prior to the very first data packet), then packet length is encoded by a single byte (`0x01-0x7e` = data length divided by 4; or `0x7f` followed by 3 bytes (little endian) divided into 4) followed by the data themselves (sequence number and CRC32 not added). In this case, server responses have the same form (although the server does not send `0xef` as the first byte).

Detailed documentation on creating authorization keys is available here ».

1. Request for (p,q) Authorization

```
req_pq#60469778 nonce:int128 = ResPQ
```

Parameter	Offset, Length in bytes	Value	Description
auth_key_id	0, 8	`0`	Since message is in plain text
message_id	8, 8	`51e57ac42770964a`	Exact unixtime * 2^32
message_length	16, 4	`20`	Message body length
%(req_pq)	20, 4	`60469778`	req_pq constructor number from TL schema
nonce	24, 16	`3E0549828CCA27E966B301A48FECE2FC`	Random number

The header is 20 bytes long, the message body is 20 bytes long, and the entire message is 40 bytes in length.

```
0000 | 00 00 00 00 00 00 00 00 4A 96 70 27 C4 7A E5 51
0010 | 14 00 00 00 78 97 46 60 3E 05 49 82 8C CA 27 E9
0020 | 66 B3 01 A4 8F EC E2 FC
```

2. A response from the server has been received with the following content:

```
0000 | 00 00 00 00 00 00 00 00 01 C8 83 1E C9 7A E5 51
0010 | 40 00 00 00 63 24 16 05 3E 05 49 82 8C CA 27 E9
0020 | 66 B3 01 A4 8F EC E2 FC A5 CF 4D 33 F4 A1 1E A8
0030 | 77 BA 4A A5 73 90 73 30 08 17 ED 48 94 1A 08 F9
0040 | 81 00 00 00 15 C4 B5 1C 01 00 00 00 21 6B E8 6C
0050 | 02 2B B4 C3
```

Response decomposition using the following formula:

```
resPQ#05162463 nonce:int128 server_nonce:int128 pq:string server_public_key_fingerprints:Vector long = ResPQ
```

Parameter	Offset, Length in bytes	Value	Description
auth_key_id	0, 8	`0`	Since message is in plain text
message_id	8, 8	`51E57AC91E83C801`	Server message ID
message_length	16, 4	`64`	Message body length
%(resPQ)	20, 4	`05162463`	resPQ constructor number from TL schema
nonce	24, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	40, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Server-generated random number
pq	56, 12	`17ED48941A08F981`	Single-byte prefix denoting length, an 8-byte string, and three bytes of padding
%(Vector long)	68, 4	`1cb5c415`	*Vector long* constructor number from TL schema
count	72, 4	`1`	Number of elements in key fingerprint list
fingerprints[]	76, 8	`c3b42b026ce86b21`	64 lower-order bits of SHA1 (server_public_key)

The **server_public_key** public key has been selected whose fingerprint corresponds to the only one received from the server: `c3b42b026ce86b21`.

3. Pq = 17ED48941A08F981 decomposed into 2 prime cofactors:

```
p = 494C553B
q = 53911073
```

4. encrypted_data Generation

```
p_q_inner_data#83c95aec pq:string p:string q:string nonce:int128 server_nonce:int128 new_nonce:int256 = P_Q_inn
```

Parameter	Offset, Length in bytes	Value	Description
% (p_q_inner_data)	0, 4	`83c95aec`	p_q_inner_data constructor number from TL schema
pq	4, 12	`17ED48941A08F981`	Single-byte prefix denoting length, 8-byte string, and three bytes of padding
p	16, 8	`494C553B`	First prime cofactor: single-byte prefix denoting length, 4-byte string, and three bytes of padding
q	24, 8	`53911073`	Second prime cofactor: single-byte prefix denoting length, 4-byte string, and three

			bytes of padding
nonce	32, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	48, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
new_nonce	64, 32	`311C85DB234AA2640AFC4A76A735CF5B 1F0FD68BD17FA181E1229AD867CC024D`	Client-generated random number

The serialization of *P_Q_inner_data* produces some string **data**. This is followed by **encrypted_data**:

```
SHA1 (data) = DB761C27718A2305044F71F2AD951629D78B2449
RSA (data_with_hash, server_public_key) = 7BB0100A523161904D9C69FA04BC60DECFC5DD74B99995C768EB60D8716E2109BAF2D
```

The length of the final string was 256 bytes.

Request to Start Diffie–Hellman Key Exchange

```
req_DH_params#d712e4be nonce:int128 server_nonce:int128 p:string q:string public_key_fingerprint:long encrypted
```

Parameter	Offset, Length in bytes	Value	Description
auth_key_id	0, 8	`0`	Since message is in plain text
message_id	8, 8	`51e57ac917717a27`	Exact unixtime * 2^32
message_length	16, 4	`320`	Message body length
%(req_DH_params)	20, 4	`d712e4be`	req_DH_params constructor number from TL schema
nonce	24, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	40, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
p	56, 8	`494C553B`	First prime cofactor: single-byte prefix denoting length, 4-byte string, and three bytes of padding
q	64, 8	`53911073`	Second prime cofactor: single-byte prefix denoting length, 4-byte string, and three bytes of padding
public_key_fingerprint	72, 8	`c3b42b026ce86b21`	Fingerprint of public key used
encrypted_data	80, 260	See above	See "Generation of encrypted_data"

```
0000 | 00 00 00 00 00 00 00 00 27 7A 71 17 C9 7A E5 51
0010 | 40 01 00 00 BE E4 12 D7 3E 05 49 82 8C CA 27 E9
0020 | 66 B3 01 A4 8F EC E2 FC A5 CF 4D 33 F4 A1 1E A8
0030 | 77 BA 4A A5 73 90 73 30 04 49 4C 55 3B 00 00 00
0040 | 04 53 91 10 73 00 00 00 21 6B E8 6C 02 2B B4 C3
0050 | FE 00 01 00 7B B0 10 0A 52 31 61 90 4D 9C 69 FA
0060 | 04 BC 60 DE CF C5 DD 74 B9 99 95 C7 68 EB 60 D8
0070 | 71 6E 21 09 BA F2 D4 60 1D AB 6B 09 61 0D C1 10
0080 | 67 BB 89 02 1E 09 47 1F CF A5 2D BD 0F 23 20 4A
0090 | D8 CA 8B 01 2B F4 0A 11 2F 44 69 5A B6 C2 66 95
00A0 | 53 86 11 4E F5 21 1E 63 72 22 7A DB D3 49 95 D3
00B0 | E0 E5 FF 02 EC 63 A4 3F 99 26 87 89 62 F7 C5 70
00C0 | E6 A6 E7 8B F8 36 6A F9 17 A5 27 26 75 C4 60 64
00D0 | BE 62 E3 E2 02 EF A8 B1 AD FB 1C 32 A8 98 C2 98
00E0 | 7B E2 7B 5F 31 D5 7C 9B B9 63 AB CB 73 4B 16 F6
00F0 | 52 CE DB 42 93 CB B7 C8 78 A3 A3 FF AC 9D BE A9
0100 | DF 7C 67 BC 9E 95 08 E1 11 C7 8F C4 6E 05 7F 5C
0110 | 65 AD E3 81 D9 1F EE 43 0A 6B 57 6A 99 BD F8 55
0120 | 1F DB 1B E2 B5 70 69 B1 A4 57 30 61 8F 27 42 7E
0130 | 8A 04 72 0B 49 71 EF 4A 92 15 98 3D 68 F2 83 0C
0140 | 3E AA 6E 40 38 55 62 F9 70 D3 8A 05 C9 F1 24 6D
0150 | C3 34 38 E6
```

5. A response from the server has been received with the following content:

```
0000 | 00 00 00 00 00 00 00 00 01 54 43 36 CB 7A E5 51
0010 | 78 02 00 00 5C 07 E8 D0 3E 05 49 82 8C CA 27 E9
0020 | 66 B3 01 A4 8F EC E2 FC A5 CF 4D 33 F4 A1 1E A8
0030 | 77 BA 4A A5 73 90 73 30 FE 50 02 00 28 A9 2F E2
0040 | 01 73 B3 47 A8 BB 32 4B 5F AB 26 67 C9 A8 BB CE
0050 | 64 68 D5 B5 09 A4 CB DD C1 86 24 0A C9 12 CF 70
0060 | 06 AF 89 26 DE 60 6A 2E 74 C0 49 3C AA 57 74 1E
0070 | 6C 82 45 1F 54 D3 E0 68 F5 CC C4 9B 44 44 12 4B
0080 | 96 66 FF B4 05 AA B5 64 A3 D0 1E 67 F6 E9 12 86
0090 | 7C 8D 20 D9 88 27 07 DC 33 0B 17 B4 E0 DD 57 CB
00A0 | 53 BF AA FA 9E F5 BE 76 AE 6C 1B 9B 6C 51 E2 D6
00B0 | 50 2A 47 C8 83 09 5C 46 C8 1E 3B E2 5F 62 42 7B
00C0 | 58 54 88 BB 3B F2 39 21 3B F4 8E B8 FE 34 C9 A0
00D0 | 26 CC 84 13 93 40 43 97 4D B0 35 56 63 30 38 39
00E0 | 2C EC B5 1F 94 82 4E 14 0B 98 63 77 30 A4 BE 79
00F0 | A8 F9 DA FA 39 BA E8 1E 10 95 84 9E A4 C8 34 67
0100 | C9 2A 3A 17 D9 97 81 7C 8A 7A C6 1C 3F F4 14 DA
0110 | 37 B7 D6 6E 94 9C 0A EC 85 8F 04 82 24 21 0F CC
0120 | 61 F1 1C 3A 91 0B 43 1C CB D1 04 CC CC 8D C6 D2
0130 | 9D 4A 5D 13 3B E6 39 A4 C3 2B BF F1 53 E6 3A CA
0140 | 3A C5 2F 2E 47 09 B8 AE 01 84 4B 14 2C 1E E8 9D
0150 | 07 5D 64 F6 9A 39 9F EB 04 E6 56 FE 36 75 A6 F8
0160 | F4 12 07 8F 3D 0B 58 DA 15 31 1C 1A 9F 8E 53 B3
0170 | CD 6B B5 57 2C 29 49 04 B7 26 D0 BE 33 7E 2E 21
0180 | 97 7D A2 6D D6 E3 32 70 25 1C 2C A2 9D FC C7 02
0190 | 27 F0 75 5F 84 CF DA 9A C4 B8 DD 5F 84 F1 D1 EB
01A0 | 36 BA 45 CD DC 70 44 4D 8C 21 3E 4B D8 F6 3B 8A
01B0 | B9 5A 2D 0B 41 80 DC 91 28 3D C0 63 AC FB 92 D6
01C0 | A4 E4 07 CD E7 C8 C6 96 89 F7 7A 00 74 41 D4 A6
01D0 | A8 38 4B 66 65 02 D9 B7 7F C6 8B 5B 43 CC 60 7E
01E0 | 60 A1 46 22 3E 11 0F CB 43 BC 3C 94 2E F9 81 93
01F0 | 0C DC 4A 1D 31 0C 0B 64 D5 E5 5D 30 8D 86 32 51
0200 | AB 90 50 2C 3E 46 CC 59 9E 88 6A 92 7C DA 96 3B
0210 | 9E B1 6C E6 26 03 B6 85 29 EE 98 F9 F5 20 64 19
0220 | E0 3F B4 58 EC 4B D9 45 4A A8 F6 BA 77 75 73 CC
0230 | 54 B3 28 89 5B 1D F2 5E AD 9F B4 CD 51 98 EE 02
0240 | 2B 2B 81 F3 88 D2 81 D5 E5 BC 58 01 07 CA 01 A5
0250 | 06 65 C3 2B 55 27 15 F3 35 FD 76 26 4F AD 00 DD
0260 | D5 AE 45 B9 48 32 AC 79 CE 7C 51 1D 19 4B C4 2B
0270 | 70 EF A8 50 BB 15 C2 01 2C 52 15 CA BF E9 7C E6
0280 | 6B 8D 87 34 D0 EE 75 9A 63 8A F0 13
```

Response decomposition using the following formula:

```
server_DH_params_fail#79cb045d nonce:int128 server_nonce:int128 new_nonce_hash:int128 = Server_DH_Params;
server_DH_params_ok#d0e8075c nonce:int128 server_nonce:int128 encrypted_answer:string = Server_DH_Params;
```

Parameter	Offset, Length in bytes	Value	Description

auth_key_id	0, 8	`0`	Since message is in plain text
message_id	8, 8	`51E57ACB36435401`	Exact unixtime * 2^32
message_length	16, 4	`632`	Message body length
% (server_DH_params_ok)	20, 4	`d0e8075c`	server_DH_params_ok constructor number from TL schema
nonce	24, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	40, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
encrypted_answer	56, 596	See below	See "Decomposition of encrypted_answer"

Conversion of encrypted_answer into answer:

```
encrypted_answer = 28A92FE20173B347A8BB324B5FAB2667C9A8BBCE6468D5B509A4CBDDC186240AC912CF7006AF8926DE606A2E74C0
tmp_aes_key = F011280887C7BB01DF0FC4E17830E0B91FBB8BE4B2267CB985AE25F33B527253
tmp_aes_iv = 3212D579EE35452ED23E0D0C92841AA7D31B2E9BDEF2151E80D15860311C85DB
answer = BA0D89B53E0549828CCA27E966B301A48FECE2FCA5CF4D33F4A11EA877BA4AA573907330020000000FE000100C71CAEB9C6B1C9
```

Server_DH_inner_data decomposition using the following formula:

```
server_DH_inner_data#b5890dba nonce:int128 server_nonce:int128 g:int dh_prime:string g_a:string server_time:int
```

Parameter	Offset, Length in bytes	Value	Description
% (server_DH_inner_data)	0, 4	`b5890dba`	server_DH_inner_data constructor number from TL schema
nonce	4, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	20, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
g	36, 4	`2`	Value received from server in Step 2
dh_prime	40, 260	`C71CAEB9C6B1C9048E6C522F70F13F73 980D40238E3E21C14934D037563D930F 48198A0AA7C14058229493D22530F4DB FA336F6E0AC925139543AED44CCE7C37 20FD51F69458705AC68CD4FE6B6B13AB DC9746512969328454F18FAF8C595F64 2477FE96BB2A941D5BCD1D4AC8CC4988 0708FA9B378E3C4F3A9060BEE67CF9A4 A4A695811051907E162753B56B0F6B41 0DBA74D8A84B2A14B3144E0EF1284754 FD17ED950D5965B4B9DD46582DB1178D 169C6BC465B0D6FF9CA3928FEF5B9AE4 E418FC15E83EBEA0F87FA9FF5EED7005 0DED2849F47BF959D956850CE929851F 0D8115F635B105EE2E4E15D04B2454BF 6F4FADF034B10403119CD8E3B92FCC5B`	
g_a	300, 260	`262AABA621CC4DF587DC94CF8252258C 0B9337DFB47545A49CDD5C9B8EAE7236 C6CADC40B24E88590F1CC2CC762EBF1C F11DCC0B393CAAD6CEE4EE5848001C73 ACBB1D127E4CB93072AA3D1C8151B6FB 6AA6124B7CD782EAF981BDCFCE9D7A00 E423BD9D194E8AF78EF6501F415522E4 4522281C79D906DDB79C72E9C63D83FB 2A940FF779DFB5F2FD786FB4AD71C9F0 8CF48758E534E9815F634F1E3A80A5E1 C2AF210C5AB762755AD4B2126DFA61A7 7FA9DA967D65DFD0AFB5CDF26C4D4E1A 88B180F4E0D0B45BA1484F95CB2712B5 0BF3F5968D9D55C99C0FB9FB67BFF56D 7D4481B634514FBA3488C4CDA2FC0659 990E8E868B28632875A9AA703BCDCE8F`	
server_time	560, 4	`1373993675`	Server time

6. Random number b is computed:

```
b = 6F620AFA575C9233EB4C014110A7BCAF49464F798A18A0981FEA1E05E8DA67D9681E0FD6DF0EDF0272AE3492451A84502F2EFC0DA18
```

Generation of encrypted_data

```
client_DH_inner_data#6643b654 nonce:int128 server_nonce:int128 retry_id:long g_b:string = Client_DH_Inner_Data
```

Parameter	Offset, Length in bytes	Value	Description
% (client_DH_inner_data)	0, 4	`6643b654`	client_DH_inner_data constructor number from TL schema
nonce	4, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	20, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
retry_id	36, 8	`0`	First attempt
g_b	44, 260	`73700E7BFC7AEEC828EB8E0DCC04D09A 0DD56A1B4B35F72F0B55FCE7DB7EBB72 D7C33C5D4AA59E1C74D09B01AE536B31 8CFED436AFDB15FE9EB4C70D7F0CB14E 46DBBDE9053A64304361EB358A9BB32E 9D5C2843FE87248B89C3F066A7D5876D 61657ACC52B0D81CD683B2A0FA93E8AD`	`g^b mod dh_prime`

```
AB20377877F3BC3369BBF57B10F5B589
[illegible]
79C1B9727A573CFFDCA8D23C721B135B
92E529B1CDD2F7ABD4F34DAC4BE1EEAF
60993DDE8ED45890E4F47C26F2C0B2E0
37BB502739C8824F2A99E2B1E7E41658
3417CC79A8807A4BDAC6A5E9805D4F61
86C37D66F6988C9F9C752896F3D34D25
529263FAF2670A09B2A59CE35264511F
```

The serialization of *Client_DH_Inner_Data* produces some string **data**. This is followed by **encrypted_data**:

```
data_with_hash := SHA1(data) + data + (0-15 random bytes); such that the length be divisible by 16;
AES256_ige_encrypt (data_with_hash, tmp_aes_key, tmp_aes_iv) = 928A4957D0463B525C1CC48AABAA030A256BE5C746792C84
```

The length of the final string was 336 bytes.

Request

```
set_client_DH_params#f5045f1f nonce:int128 server_nonce:int128 encrypted_data:string = Set_client_DH_params_ans
```

Parameter	Offset, Length in bytes	Value	Description
auth_key_id	0, 8	`0`	Since message is in plain text
message_id	8, 8	`51e57acd2aa32c6d`	Exact unixtime * 2^32
message_length	16, 4	`376`	Message body length
% (set_client_DH_params)	20, 4	`f5045f1f`	set_client_DH_params constructor number from TL schema
nonce	24, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	40, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
encrypted_data	56, 340	See above	See "Generation of encrypted_data"

```
0000 | 00 00 00 00 00 00 00 00 6D 2C A3 2A CD 7A E5 51
0010 | 78 01 00 00 1F 5F 04 F5 3E 05 49 82 8C CA 27 E9
0020 | 66 B3 01 A4 8F EC E2 FC A5 CF 4D 33 F4 A1 1E A8
0030 | 77 BA 4A A5 73 90 73 30 FE 50 01 00 92 8A 49 57
0040 | D0 46 3B 52 5C 1C C4 8A AB AA 03 0A 25 6B E5 C7
0050 | 46 79 2C 84 CA 4C 5A 0D F6 0A C7 99 04 8D 98 A3
0060 | 8A 84 80 ED CF 08 22 14 DF C7 9D CB 9E E3 4E 20
0070 | 65 13 E2 B3 BC 15 04 CF E6 C9 AD A4 6B F9 A0 3C
0080 | A7 4F 19 2E AF 8C 27 84 54 AD AB C7 95 A5 66 61
0090 | 54 62 D3 18 17 38 29 84 03 95 05 F7 1C B3 3A 41
00A0 | E2 52 7A 4B 1A C0 51 07 87 2F ED 8E 3A BC EE 15
00B0 | 18 AE 96 5B 0E D3 AE D7 F6 74 79 15 5B DA 8E 4C
00C0 | 28 6B 64 CD F1 23 EC 74 8C F2 89 B1 DB 02 D1 90
00D0 | 7B 56 2D F4 62 D8 58 2B A6 F0 A3 02 2D C2 D3 50
00E0 | 4D 69 D1 BA 48 B6 77 E3 A8 30 BF AF D6 75 84 C8
00F0 | AA 24 E1 34 4A 89 04 E3 05 F9 58 7C 92 EF 96 4F
0100 | 00 83 F5 0F 61 EA B4 A3 93 EA A3 3C 92 70 29 4A
0110 | ED C7 73 28 91 D4 EA 15 99 F5 23 11 D7 44 69 D2
0120 | 11 2F 4E DF 3F 34 2E 93 C8 E8 7E 81 2D C3 98 9B
0130 | AE CF E6 74 0A 46 07 75 24 C7 50 93 F5 A5 40 57
0140 | 36 DE 89 37 BB 6E 42 C9 A0 DC F2 2C A5 32 27 D4
0150 | 62 BC CC 2C FE 94 B6 FE 86 AB 7F BF A3 95 02 1F
0160 | 66 66 1A F7 C0 02 4C A2 98 6C A0 3F 34 76 90 54
0170 | 07 D1 EA 9C 01 0B 76 32 58 DB 1A A2 CC 78 26 D9
0180 | 13 34 EF C1 FD C6 65 B6 7F E4 5E D0
```

7. Computing auth_key using formula `g^{ab} mod dh_prime`:

```
auth_key = AB96E207C631300986F30EF97DF55E179E63C112675F0CE502EE76D74BBEE6CBD1E95772818881E9F2FF54BD52C258787474
```

8. The server verifies that auth_key_hash is unique.

The key is unique.

9. A response from the server has been received with the following content:

```
0000 | 00 00 00 00 00 00 00 00 01 30 AA C5 CE 7A E5 51
0010 | 34 00 00 00 34 F7 CB 3B 3E 05 49 82 8C CA 27 E9
0020 | 66 B3 01 A4 8F EC E2 FC A5 CF 4D 33 F4 A1 1E A8
0030 | 77 BA 4A A5 73 90 73 30 CC EB C0 21 72 66 E1 ED
0040 | EC 7F B0 A0 EE D6 C2 20
```

Set_client_DH_params_answer decomposition using the following formula:

```
dh_gen_ok#3bcbf734 nonce:int128 server_nonce:int128 new_nonce_hash1:int128 = Set_client_DH_params_answer;
dh_gen_retry#46dc1fb9 nonce:int128 server_nonce:int128 new_nonce_hash2:int128 = Set_client_DH_params_answer;
dh_gen_fail#a69dae02 nonce:int128 server_nonce:int128 new_nonce_hash3:int128 = Set_client_DH_params_answer;
```

Parameter	Offset, Length in bytes	Value	Description
%(dh_gen_ok)	0, 4	`3bcbf734`	dh_gen_ok constructor number from TL schema
nonce	4, 16	`3E0549828CCA27E966B301A48FECE2FC`	Value generated by client in Step 1
server_nonce	20, 16	`A5CF4D33F4A11EA877BA4AA573907330`	Value received from server in Step 2
new_nonce_hash1	36, 16	`CCEBC0217266E1EDEC7FB0A0EED6C220`	

Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About
FAQ
Blog
Jobs

Mobile Apps
iPhone/iPad
Android
Windows Phone

Desktop Apps
PC/Mac/Linux
macOS
Web-browser

Platform
API
Translations
Instant View

Service Messages

Response to an RPC query

A response to an RPC query is normally wrapped as follows:

```
rpc_result#f35c6d01 req_msg_id:long result:Object = RpcResult;
```

Here req_msg_id is the identifier of the message sent by the other party and containing an RPC query. This way, the recipient knows that the result is a response to the specific RPC query in question.
At the same time, this response serves as acknowledgment of the other party's receipt of the req_msg_id message.

Note that the response to an RPC query must also be acknowledged. Most frequently, this coincides with the transmission of the next message (which may have a container attached to it carrying a service message with the acknowledgment).

RPC Error

The result field returned in response to any RPC query may also contain an error message in the following format:

```
rpc_error#2144ca19 error_code:int error_message:string = RpcError;
```

Cancellation of an RPC Query

In certain situations, the client does not want to receive a response to an already transmitted RPC query, for example because the response turns out to be long and the client has decided to do without it because of insufficient link capacity. Simply interrupting the TCP connection will not have any effect because the server would re-send the missing response at the first opportunity. Therefore, the client needs a way to cancel receipt of the RPC response message, actually acknowledging its receipt prior to it being in fact received, which will settle the server down and prevent it from re-sending the response. However, the client does not know the RPC response's msg_id prior to receiving the response; the only thing it knows is the req_msg_id. i. e. the msg_id of the relevant RPC query. Therefore, a special query is used:

```
rpc_drop_answer#58e4a740 req_msg_id:long = RpcDropAnswer;
```

The response to this query returns as one of the following messages wrapped in rpc_result and requiring an acknowledgment:

```
rpc_answer_unknown#5e2ad36e = RpcDropAnswer;
rpc_answer_dropped_running#cd78e586 = RpcDropAnswer;
rpc_answer_dropped#a43ad8b7 msg_id:long seq_no:int bytes:int = RpcDropAnswer;
```

The first version of the response is used if the server remembers nothing of the incoming req_msg_id (if it has already been responded to, for example). The second version is used if the response was canceled while the RPC query was being processed (where the RPC query itself was still fully processed); in this case, the same rpc_answer_dropped_running is also returned in response to the original query, and both of these responses require an acknowledgment from the client. The final version means that the RPC response was removed from the server's outgoing queue, and its msg_id, seq_no, and length in bytes are transmitted to the client.

Note that rpc_answer_dropped_running and rpc_answer_dropped serve as acknowledgments of the server's receipt of the original query (the same one, the response to which we wish to forget). In addition, same as for any RPC queries, any response to rpc_drop_answer is an acknowledgment for rpc_drop_answer itself.

As an alternative to using rpc_drop_answer, a new session may be created after the connection is reset and the old session is removed through destroy_session.

Messages associated with querying, changing, and receiving the status of other messages

See Mobile Protocol: Service Messages about Messages

Request for several future salts

The client may at any time request from the server several (between 1 and 64) future server salts together with their validity periods. Having stored them in persistent memory, the client may use them to send messages in the future even if he changes sessions (a server salt is attached to the authorization key rather than being session-specific).

```
get_future_salts#b921bd04 num:int = FutureSalts;
future_salt#0949d9dc valid_since:int valid_until:int salt:long = FutureSalt;
future_salts#ae500895 req_msg_id:long now:int salts:vector future_salt = FutureSalts;
```

The client must check to see that the response's req_msg_id in fact coincides with msg_id of the query for get_future_salts. The server returns a maximum of num future server salts (may return fewer). The response serves as the acknowledgment of the query and does not require an acknowledgment itself.

Ping Messages (PING/PONG)

```
ping#7abe77ec ping_id:long = Pong;
```

A response is usually returned to the same connection:

```
pong#347773c5 msg_id:long ping_id:long = Pong;
```

These messages do not require acknowledgments. A pong is transmitted only in response to a ping while a ping can be initiated by either side.

Deferred Connection Closure + PING

```
ping_delay_disconnect#f3427b8c ping_id:long disconnect_delay:int = Pong;
```

Works like ping. In addition, after this is received, the server starts a timer which will close the current connection disconnect_delay seconds later unless it receives a new message of the same type which automatically resets all previous timers. If the client sends these pings once every 60 seconds, for example, it may set disconnect_delay equal to 75 seconds.

Request to Destroy Session

Used by the client to notify the server that it may forget the data from a different session belonging to the same user (i. e. with the same auth_key_id). The result of this being applied to the current session is undefined.

```
destroy_session#e7512126 session_id:long = DestroySessionRes;
destroy_session_ok#e22045fc session_id:long = DestroySessionRes;
destroy_session_none#62d350c9 session_id:long = DestroySessionRes;
```

New Session Creation Notification

The server notifies the client that a new session (from the server's standpoint) had to be created to handle a client message. If, after this, the server receives a message with an even smaller msg_id within the same session, a similar notification will be generated for this msg_id as well. No such notifications are generated for high msg_id values.

```
new_session_created#9ec20908 first_msg_id:long unique_id:long server_salt:long = NewSession
```

The unique_id parameter is generated by the server every time a session is (re-)created.

This notification must be acknowledged by the client. It is necessary, for instance, for the client to understand that there is, in fact, a "gap" in the stream of long poll notifications received from the server (the user may have failed to

receive notifications during some period of time).

Notice that the server may unilaterally destroy (close) any existing client sessions with all pending messages and notifications, without sending any notifications. This happens, for example, if the session is inactive for a long time, and the server runs out of memory. If the client at some point decides to send new messages to the server using the old session, already forgotten by the server, such a "new session created" notification will be generated. The client is expected to handle such situations gracefully.

Containers

Containers are messages containing several other messages. Used for the ability to transmit several RPC queries and/or service messages at the same time, using HTTP or even TCP or UDP protocol. A container may only be accepted or rejected by the other party as a whole.

Simple Container

A simple container carries several messages as follows:

```
msg_container#73f1f8dc messages:vector message = MessageContainer;
```

Here message refers to any message together with its length and msg_id:

```
message msg_id:long seqno:int bytes:int body:Object = Message;
```

`bytes` is the number of bytes in the body serialization.
All messages in a container must have msg_id lower than that of the container itself. A container does not require an acknowledgment and may not carry other simple containers. When messages are re-sent, they may be combined into a container in a different manner or sent individually.

Empty containers are also allowed. They are used by the server, for example, to respond to an HTTP request when the timeout specified in http_wait expires, and there are no messages to transmit.

Message Copies

In some situations, an old message with a msg_id that is no longer valid needs to be re-sent. Then, it is wrapped in a copy container:

```
msg_copy#e06046b2 orig_message:Message = MessageCopy;
```

Once received, the message is processed as if the wrapper were not there. However, if it is known for certain that the message orig_message.msg_id was received, then the new message is not processed (while at the same time, it and orig_message.msg_id are acknowledged). The value of orig_message.msg_id must be lower than the container's msg_id.

This is not used at this time, because an old message can be wrapped in a simple container with the same result.

Packed Object

Used to replace any other object (or rather, a serialization thereof) with its archived (gzipped) representation:

```
gzip_packed#3072cfa1 packed_data:string = Object;
```

At the present time, it is supported in the body of an RPC response (i.e., as result in rpc_result) and generated by the server for a limited number of high-level queries. In addition, in the future it may be used to transmit non-service messages (i. e. RPC queries) from client to server.

HTTP Wait/Long Poll

The following special service query not requiring an acknowledgement (which must be transmitted only through an HTTP connection) is used to enable the server to send messages in the future to the client using HTTP protocol:

```
http_wait#9299359f max_delay:int wait_after:int max_wait:int = HttpWait;
```

When such a message (or a container carrying such a message) is received, the server either waits `max_delay` milliseconds, whereupon it forwards all the messages that it is holding on to the client if there is at least one message queued in session (if needed, by placing them into a container to which acknowledgments may also be added); or else waits no more than `max_wait` milliseconds until such a message is available. If a message never appears, an empty container is transmitted.

The `max_delay` parameter denotes the maximum number of milliseconds that has elapsed between the first message for this session and the transmission of an HTTP response. The `wait_after` parameter works as follows: after the receipt of the latest message for a particular session, the server waits another `wait_after` milliseconds in case there are more messages. If there are no additional messages, the result is transmitted (a container with all the messages). If more messages appear, the `wait_after` timer is reset.

At the same time, the `max_delay` parameter has higher priority than `wait_after`, and `max_wait` has higher priority than `max_delay`.

This message does not require a response or an acknowledgement. If the container transmitted over HTTP carries several such messages, the behavior is undefined (in fact, the latest parameter will be used).

If no `http_wait` is present in container, default values `max_delay=0` (milliseconds), `wait_after=0` (milliseconds), and `max_wait=25000` (milliseconds) are used.

If the client's ping of the server takes a long time, it may make sense to set `max_delay` to a value that is comparable in magnitude to ping time.

Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About
FAQ
Blog
Jobs

Mobile Apps
iPhone/iPad
Android
Windows Phone

Desktop Apps
PC/Mac/Linux
macOS
Web-browser

Platform
API
Translations
Instant View

Service Messages about Messages

Acknowledgment of Receipt

Receipt of virtually all messages (with the exception of some purely service ones as well as the plain-text messages used in the protocol for creating an authorization key) must be acknowledged.
This requires the use of the following service message (not requiring an acknowledgment):

```
msgs_ack#62d6b459 msg_ids:Vector long = MsgsAck;
```

A server usually acknowledges the receipt of a message from a client (normally, an RPC query) using an RPC response. If a response is a long time coming, a server may first send a receipt acknowledgment, and somewhat later, the RPC response itself.

A client normally acknowledges the receipt of a message from a server (usually, an RPC response) by adding an acknowledgment to the next RPC query if it is not transmitted too late (if it is generated, say, 60–120 seconds following the receipt of a message from the server). However, if for a long period of time there is no reason to send messages to the server or if there is a large number of unacknowledged messages from the server (say, over 16), the client transmits a stand-alone acknowledgment.

Notice of Ignored Error Message

In certain cases, a server may notify a client that its incoming message was ignored for whatever reason. Note that such a notification cannot be generated unless a message is correctly decoded by the server.

```
bad_msg_notification#a7eff811 bad_msg_id:long bad_msg_seqno:int error_code:int = BadMsgNotification;
bad_server_salt#edab447b bad_msg_id:long bad_msg_seqno:int error_code:int new_server_salt:long = BadMsgNotifica
```

Here, error_code can also take on the following values:

- 16: msg_id too low (most likely, client time is wrong; it would be worthwhile to synchronize it using msg_id notifications and re-send the original message with the "correct" msg_id or wrap it in a container with a new msg_id if the original message had waited too long on the client to be transmitted)
- 17: msg_id too high (similar to the previous case, the client time has to be synchronized, and the message re-sent with the correct msg_id)
- 18: incorrect two lower order msg_id bits (the server expects client message msg_id to be divisible by 4)
- 19: container msg_id is the same as msg_id of a previously received message (this must never happen)
- 20: message too old, and it cannot be verified whether the server has received a message with this msg_id or not
- 32: msg_seqno too low (the server has already received a message with a lower msg_id but with either a higher or an equal and odd seqno)
- 33: msg_seqno too high (similarly, there is a message with a higher msg_id but with either a lower or an equal and odd seqno)
- 34: an even msg_seqno expected (irrelevant message), but odd received
- 35: odd msg_seqno expected (relevant message), but even received
- 48: incorrect server salt (in this case, the bad_server_salt response is received with the correct salt, and the message is to be re-sent with it)
- 64: invalid container.

The intention is that error_code values are grouped (error_code >> 4): for example, the codes 0x40 – 0x4f correspond to errors in container decomposition.

Notifications of an ignored message do not require acknowledgment (i.e., are irrelevant).

Important: if server_salt has changed on the server or if client time is incorrect, any query will result in a notification in the above format. The client must check that it has, in fact, recently sent a message with the specified msg_id, and if that is the case, update its time correction value (the difference between the client's and the server's clocks) and the server salt based on msg_id and the server_salt notification, so as to use these to (re)send future messages. In the meantime, the original message (the one that caused the error message to be returned) must also be re-sent with a better msg_id and/or server_salt.

In addition, the client can update the server_salt value used to send messages to the server, based on the values of RPC responses or containers carrying an RPC response, provided that this RPC response is actually a match for the query sent recently. (If there is doubt, it is best not to update since there is risk of a replay attack).

Request for Message Status Information

If either party has not received information on the status of its outgoing messages for a while, it may explicitly request it from the other party:

```
msgs_state_req#da69fb52 msg_ids:Vector long = MsgsStateReq;
```

The response to the query contains the following information:

Informational Message regarding Status of Messages

```
msgs_state_info#04deb57d req_msg_id:long info:string = MsgsStateInfo;
```

Here, `info` is a string that contains exactly one byte of message status for each message from the incoming msg_ids list:

- 1 = nothing is known about the message (msg_id too low, the other party may have forgotten it)
- 2 = message not received (msg_id falls within the range of stored identifiers; however, the other party has certainly not received a message like that)
- 3 = message not received (msg_id too high; however, the other party has certainly not received it yet)
- 4 = message received (note that this response is also at the same time a receipt acknowledgment)
- +8 = message already acknowledged
- +16 = message not requiring acknowledgment
- +32 = RPC query contained in message being processed or processing already complete
- +64 = content-related response to message already generated
- +128 = other party knows for a fact that message is already received

This response does not require an acknowledgment. It is an acknowledgment of the relevant msgs_state_req, in and of itself.

Note that if it turns out suddenly that the other party does not have a message that looks like it has been sent to it, the message can simply be re-sent. Even if the other party should receive two copies of the message at the same time, the duplicate will be ignored. (If too much time has passed, and the original msg_id is not longer valid, the message is to be wrapped in msg_copy).

Voluntary Communication of Status of Messages

Either party may voluntarily inform the other party of the status of the messages transmitted by the other party.

```
msgs_all_info#8cc0d131 msg_ids:Vector long info:string = MsgsAllInfo
```

All message codes known to this party are enumerated, with the exception of those for which the +128 and the +16 flags are set. However, if the +32 flag is set but not +64, then the message status will still be communicated.

This message does not require an acknowledgment.

Extended Voluntary Communication of Status of One Message

Normally used by the server to respond to the receipt of a duplicate msg_id, especially if a response to the message has already been generated and the response is large. If the response is small, the server may re-send the answer itself instead. This message can also be used as a notification instead of resending a large message.

```
msg_detailed_info#276d3ec6 msg_id:long answer_msg_id:long bytes:int status:int = MsgDetailedInfo;
msg_new_detailed_info#809db6df answer_msg_id:long bytes:int status:int = MsgDetailedInfo;
```

The second version is used to notify of messages that were created on the server not in response to an RPC query (such as notifications of new messages) and were transmitted to the client some time ago, but not acknowledged.

Currently, `status` is always zero. This may change in future.

This message does not require an acknowledgment.

Explicit Request to Re-Send Messages

```
msg_resend_req#7d861a08 msg_ids:Vector long = MsgResendReq;
```

The remote party immediately responds by re-sending the requested messages, normally using the same connection that was used to transmit the query. If at least one message with requested msg_id does not exist or has already been forgotten, or has been sent by the requesting party (known from parity), MsgsStateInfo is returned for all messages requested as if the MsgResendReq query had been a MsgsStateReq query as well.

Explicit Request to Re-Send Answers

```
msg_resend_ans_req#8610baeb msg_ids:Vector long = MsgResendReq;
```

The remote party immediately responds by re-sending *answers* to the requested messages, normally using the same connection that was used to transmit the query. `MsgsStateInfo` is returned for all messages requested as if the MsgResendReq query had been a `MsgsStateReq` query as well.

Binary Data Serialization

MTProto operation requires that elementary and composite data types as well as queries to which such data types are passed as arguments or by which they are returned, be transmitted in binary format (i. e. *serialized*) .
The TL language is used to describe the data types to be serialized.

General Definitions

For our purposes, we can identify a *type* with the set of its *(serialized) values* understood as strings (finite sequences) of 32-bit numbers (transmitted in little endian order).

Therefore:

- *Alphabet* (A), in this case, is a set of 32-bit numbers (normally, signed, i. e. between -2^31 and 2^31 - 1).
- *Value*, in this case, is the same as a *string in Alphabet A*, i. e. a finite (possibly, empty) sequence of 32-bit numbers. The set of all such sequences is designated as *A**.
- *Type*, for our purposes, is the same as the set of legal values of a type, i. e. some set T which is a subset of A* and is a prefix code (i. e. no element of T may be a prefix for any other element). Therefore, any sequence from A* can contain no more than one prefix that is a member of T.
- *Value of Type T* is any sequence (value) which is a member of T as a subset of A*.
- *Compatible Types* are the types T and T' not intersecting as subsets of A*, such that the union of T and T' is a prefix code.
- *Coordinated System of Types* is a finite or infinite set of types T_1, ..., T_n, ..., such that any two types from this set are compatible.
- *Data Type* is the same as *type* in the sense of the definition above.
- *Functional Type* is a type describing a function; it is not a type in the sense of the definition above. Initially, we ignore the existence of functional types and describe only the data types; however, in reality, functional types will later be implemented in some extension of this system using the so-called *temporary combinators*.

Combinators, Constructors, Composite Data Types

- *Combinator* is a function that takes arguments of certain types and returns a value of some other type. We normally look at combinators whose argument and result types are data types (rather than functional types).
- *Arity (of combinator)* is a non-negative integer, the number of combinator arguments.
- *Combinator identifier* is an identifier beginning with a lowercase Roman letter that uniquely identifies a combinator.
- *Combinator number* or *combinator name* is a 32-bit number (i.e., an element of A) that uniquely identifies a combinator. Most often, it is CRC32 of the string containing the combinator description without the final semicolon, and with one space between contiguous lexemes. This always falls in the range from 0x01000000 to 0xffffff00. The highest 256 values are reserved for the so-called *temporal-logic combinators* used to transmit functions. We frequently denote as *combinator* the combinator name with single quotes: '*combinator*'.
- *Combinator description* is a string of format `combinator_name type_arg_1 ... type_arg_N = type_res;` where `N` stands for the arity of the combinator, `type_arg_i` is the type of the i-th argument (or rather, a string with the combinator name), and `type_res` is the combinator value type.
- *Constructor* is a combinator that cannot be computed (reduced). This is used to represent composite data types. For example, combinator 'int_tree' with description `int_tree IntTree int IntTree = IntTree`, alongside combinator `empty_tree = IntTree`, may be used to define a composite data type called "IntTree" that takes on values in the form of binary trees with integers as nodes.
- *Function (functional combinator)* is a combinator which may be computed (reduced) on condition that the requisite number of arguments of requisite types are provided. The result of the computation is an expression consisting of constructors and base type values only.
- *Normal form* is an expression consisting only of constructors and base type values; that which is normally the result of computing a function.
- *Type identifier* is an identifier that normally starts with a capital letter in Roman script and uniquely identifies the type.
- *Type number* or *type name* is a 32-bit number that uniquely identifies a type; it normally is the sum of the CRC32 values of the descriptions of the type constructors.
- *Description of (composite) Type T* is a collection of the descriptions of all constructors that take on Type *T* values. This is normally written as text with each string containing the description of a single constructor. Here is a description of Type 'IntTree', for example:

  int_tree IntTree int IntTree = IntTree;
  empty_tree = IntTree;

- *Polymorphic type* is a type whose description contains parameters (*type variables*) in lieu of actual types; approximately, what would be a template in C++. Here is a description of Type `List alpha` where `List` is a polymorphic type of arity 1 (i. e., dependent on a single argument), and `alpha` is a type variable which appears as the constructor's optional parameter (in curly braces):

  cons {alpha:Type} alpha (List alpha) = List alpha;
  nil {alpha:Type} = List alpha;

- *Value of (composite) Type T* is any sequence from A* in the format `constr_num arg1 ... argN`, where constr_num is the index number of some Constructor *C* which takes on values of Type *T*, and arg_i is a value of Type *T_i* which is the type of the i-th argument to Constructor *C*. For example, let Combinator int_tree have the index number 17, whereas Combinator empty_tree has the index number 239. Then, the value of Type `IntTree` is, for example, `17 17 239 1 239 2 239` which is more conveniently written as `'int_tree' 'int_tree' 'empty_tree' 1 'empty_tree' 2 'empty_tree'`. From the standpoint of a high-level language, this is `int_tree (int_tree (empty_tree) 1 (empty_tree)) 2 (empty_tree): IntTree`.
- *Schema* is a collection of all the (composite) data type descriptions. This is used to define some agreed-to system of types.

Boxed and Bare Types

- *Boxed type* is a type any value of which starts with the constructor number. Since every constructor has a uniquely determined value type, the first number in any boxed type value uniquely defines its type. This guarantees that the various boxed types in totality make up a coordinated system of types. A boxed type identifier is always capitalized.
- *Bare type* is a type whose values do not contain a constructor number, which is implied instead. A bare type identifier always coincides with the name of the implied constructor (and therefore, begins with a lowercase letter) which may be padded at the front by the percentage sign (%). In addition, if `X` is a boxed type with no more than a single constructor, then `%X` refers to the corresponding bare type. The values of a bare type are identical with the set of number sequences obtained by dropping the first number (i. e., the external constructor index number) from the set of values of the corresponding boxed type (which is the result type of the selected constructor), starting with the selected constructor index number. For example, `3 4` is a value of the `int_couple` bare type, defined using `int_couple int int = IntCouple`. The corresponding boxed type is `IntCouple`; if 404 is the constructor index number for `int_couple`, then `404 3 4` is the value for the `IntCouple` boxed type which corresponds to the value of the bare type `int_couple` (also known as `%int_couple` and `%IntCouple`; the latter form is conceptually preferable but longer).

Conceptually, only boxed types should be used everywhere. However, for speed and compactness, bare types have to be used (for instance, an array of 10,000 bare int values is 40,000 bytes long, whereas boxed Int values take up twice as much space; therefore, when transmitting a large array of integer identifiers, say, it is more efficient to use the `Vector int` type rather than `Vector Int`). In addition, all base types (int, long, double, string) are bare.

If a boxed type is polymorphic of type arity r, this is also true of any derived bare type. In other words, if one were to define `intCouple {alpha:Type} int alpha = IntCouple alpha`, then, thereafter, intCouple as an identifier would also be a polymorphic type of arity 1 in combinator (and consequently, in constructor and type) descriptions. The notations `intCouple X`, `%(IntCouple X)`, and `%IntCouple X` are equivalent.

Base Types

Base types exist both as bare (int, long, double, string) and as boxed (Int, Long, Double, String) versions. Their *constructor* identifiers coincide with the names of the relevant bare types. Their pseudodescriptions have the following appearance:

```
int ? = Int;
long ? = Long;
double ? = Double;
string ? = String;
```

Consequently, the `int` constructor index number, for example, is the CRC32 of the string `"int ? = Int"`.

The values of bare type `int` are exactly all the single-element *sequences*, i. e. numbers between -2^31 and 2^31-1 represent themselves in this case. Values of type `long` are two-element sequences that are 64-bit signed numbers (little endian again). Values of type `double`, again, are two-element sequences containing 64-bit real numbers in a standard double format. And finally, the values of type `string` look differently depending on the length L of the string being serialized:

- If L <= 253, the serialization contains one byte with the value of L, then L bytes of the string followed by 0 to 3 characters containing 0, such that the overall length of the value be divisible by 4, whereupon all of this is interpreted as a sequence of int(L/4)+1 32-bit numbers.
- If L >= 254, the serialization contains byte 254, followed by 3 bytes with the string length L, followed by L bytes of the string, further followed by 0 to 3 null padding bytes.

Object Pseudotype

The `Object` pseudotype is a "type" which can take on values that belong to any boxed type in the schema. This helps quickly define such types as *list of random items* without using polymorphic types. It is best not to abuse this capability since it results in the use of dynamic typing. Nonetheless, it is hard to imagine the data structures that we know from PHP and JSON without using the Object pseudotype.

It is recommended to use `TypedObject` instead whenever possible:

```
object X:Type value:X = TypedObject;
```

Built-In Composite Types: Vectors and Associative Arrays

The Vector t polymorphic pseudotype is a "type" whose value is a sequence of values of any type t, either boxed or bare.

```
vector {t:Type} # [ t ] = Vector t;
```

Serialization always uses the same constructor "vector" (const 0x1cb5c415 = crc32("vector t:Type # [t] = Vector t") that is not dependent on the specific value of the variable of type t. The value of the Vector t type is the index number of the relevant constructor number followed by N, the number of elements in the vector, and then by N values of type t. The value of the optional parameter t is not involved in the serialization since it is derived from the result type (always known prior to deserialization).

Polymorphic pseudotypes IntHash t and StrHash t are associative arrays mapping integer and string keys to values of type t. They are, in fact, vectors containing bare 2-tuples (int, t) or (string, t):

```
coupleInt {t:Type} int t = CoupleInt t;
intHash {t:Type} (vector %(CoupleInt t)) = IntHash t;
coupleStr {t:Type} string t = CoupleStr t;
strHash {t:Type} (vector %(CoupleStr t)) = StrHash t;
```

The percentage sign, in this case, means that a bare type that corresponds to the boxed type in parentheses is taken; the boxed type in question must have no more than a single constructor, whatever the values of the parameters.

The keys may be sorted or be in some other order (as in PHP arrays). For associative arrays with sorted keys, the IntSortedHash or StrSortedHash alias is used:

```
intSortedHash {t:Type} (intHash t) = IntSortedHash t;
strSortedHash {t:Type} (strHash t) = StrSortedHash t;
```

Polymorphic Type Constructors

The constructor of a polymorphic type does not depend on the specific types to which the polymorphic type is applied. When it is computed, optional parameters (normally containing type variables and placed in curly braces) cease to be optional (the curly braces are removed), and, in addition to that, all parenthesis are also removed. Therefore,

```
vector {t:Type} # [ t ] = Vector t;
```

corresponds to the constructor number crc32("vector t:Type # [t] = Vector t") = 0x1cb5c415. During (de)serialization, the specific values of the optional variable t are derived from the result type (i. e. the object being serialized or deserialized) that is always known, and are never serialized explicitly.

Previously, it had to be known which specific variable types each polymorphic type will apply to. To accomplish this, the type system used strings of the form

```
polymorphic_type_name type_1 ... type_N;
```

For example,

```
Vector int;
Vector string;
Vector Object;
```

Now they are ignored.

See also polymorphism in TL.

In this case, the Object pseudotype permits using Vector Object to store lists of anything (the values of any boxed types). Since bare types are efficient when short, in practice it is unlikely that cases more complex than the ones cited above will be required.

Field Names

Let us say that we need to represent *users* as triplets containing one integer (user ID) and two strings (first and last names). The requisite data structure is the triplet int, string, string which may be declared as follows:

```
user int string string = User;
```

On the other hand, a group may be described by a similar triplet consisting of a group ID, its name, and description:

```
group int string string = Group;
```

For the difference between User and Group to be clear, it is convenient to assign names to some or all of the fields:

```
user id:int first_name:string last_name:string = User;
group id:int title:string description:string = Group;
```

If the User type needs to be extended at a later time by having records with some additional field added to it, it could be accomplished as follows:

```
userv2 id:int unread_messages:int first_name:string last_name:string in_groups:vector int = User;
```

Aside from other things, this approach helps define correct mappings between fields that belong to different constructors of the same type, convert between them as well as convert type values into an associative array with string keys (field names, if defined, are natural choices for such keys).

TL Language

See TL Language

Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed.

About

FAQ
Blog
Jobs

Mobile Apps

iPhone/iPad
Android
Windows Phone

Desktop Apps

PC/Mac/Linux
macOS
Web-browser

Platform

API
Translations
Instant View

TL Language

TL (Type Language) serves to describe the used system of types, constructors, and existing functions. In fact, the combinator description format presented in Binary Data Serialization is used.

See also:
Polymorphism in TL

Advanced topics:
Dependent types in TL
Formal description of TL
Formal description of TL combinators
Type serialization
TL schema for serialization of TL schemas

Optional combinator parameters and their values
Binary serialization and abstract TL types
Formal description of templates in TL

Overview

A TL program usually consists of two sections separated by keyword `---functions---`. The first section consists of declarations of built-in types and aggregate types (i.e. their constructors). The second section consists of the declared functions, i.e. functional combinators.

Actually, both the first and second sections consist of combinator declarations, each of which ends with a semicolon. However, the first section contains only constructors, while the second section only involves functions. Each combinator is declared using a "combinator declaration" in the format explained above. However, the combinator number and field names may be explicitly assigned.

If additional type declarations are required after functions have been declared, the keyword (section divider) `---types---` is used. Furthermore, a functional combinator may be declared in the type section if its result type begins with an exclamation point (in fact, when the function section is interpreted, this exclamation point is added automatically).

To explicitly define 32-bit names of combinators, a hash mark (#) is added immediately after the combinator's name, followed by 8 hexadecimal digits.

Namespaces

Composite constructions like `<namespace_identifier>.<constructor_identifier>` and `<namespace_identifier>.<Type_identifier>` can be used as constructor- or type identifiers. The portion of the identifier to the left of the period is called the *namespace*. Moreover, the rule about a first uppercase letter in type identifiers and lowercase letter in constructor identifiers applies to the part of the construction after the period. For example, `auth.Message` would be a type, while `auth.std_message` would be a constructor.

Namespaces do not require a special declaration.

Comments

Comments are the same as in C++.

Example

```
// built-in types
int#a8509bda ? = Int;
long ? = Long;
double ? = Double;
string ? = String;
null = Null;

vector {t:Type} # [ t ] = Vector t;
coupleInt {alpha:Type} int alpha = CoupleInt<alpha>;
coupleStr {gamma:Type} string gamma = CoupleStr gamma;
/* The name of the type variable is irrelevant: "gamma" could be replaced with "alpha";
   However, the combinator number will depend on the specific choice. */

intHash {alpha:Type} vector<coupleInt<alpha>> = IntHash<alpha>;
strHash {alpha:Type} (vector (coupleStr alpha)) = StrHash alpha;
intSortedHash {alpha:Type} intHash<alpha> = IntSortedHash<alpha>;
strSortedHash {alpha:Type} (strHash alpha) = StrSortedHash alpha;

// custom types
pair x:Object y:Object = Pair;
triple x:Object y:Object z:Object = Triple;

user#d23c81a3 id:int first_name:string last_name:string = User;
no_user#c67599d1 id:int = User;
group id:int title:string last_name:string = Group;
no_group = Group;

---functions---

// Maybe some built-in arithmetic functions; inverse quotes make "identifiers" out of arbitrary non-alphanumeri
`+` Int Int = Int;
`-` Int Int = Int;
`+` Double Double = Double;
// ...

// API functions (aka RPC functions)
getUser#b0f732d5 int = User;
getUsers#2d84d5f5 (Vector int) = Vector User;
```

In this case, the `user` constructor has been explicitly assigned a number (0xd23c81a3); In fact, this was not necessary, since this value is the CRC32 of the string `"user id:int first_name:string last_name:string = User"`, which would have been used by default.

Special constructors are not required for Vector int, Vector User, Vector Object, etc. — the same universal constructor can be used everywhere:

```
vector#1cb5c415 {t:Type} # [ t ] = Vector t;
```

Note that when the `getUsers (Vector int) = Vector User;` constructor number is calculated, the CRC32 of the string "getUsers Vector int = Vector User" is computed (from which all parentheses have been removed).

Notation `T0<T1,T2,...,Tn>` is syntactic sugar for `(T0 (T1) (T2) ... (Tn))`. For example, `Vector<User>` and `(Vector User)` are entirely interchangeable.

Example of an RPC query

Suppose we want to call `getUsers([2,3,4])`. This query will be serialized into a sequence of 32-bit integers as follows:

```
0x2d84d5f5 0x1cb5c415 0x3 0x2 0x3 0x4
```

Please note that TL serialization yields sequences of 32-bit integers. When it has to be embedded into a byte stream, for example a network packet, each 32-bit integer is represented by four bytes in little-endian order. In this way the above query corresponds to the following byte stream:

```
F5 D5 84 2D 15 C4 B5 1C 03 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00
```

The response might look something like this:

```
0x1cb5c415 0x3 0xd23c81a3 0x2 0x74655005 0x00007265 0x72615006 0x72656b 0xc67599d1 0x3 0xd23c81a3 0x4 0x686f4a0
```

This roughly corresponds to

```
[{"id":2,"first_name":"Peter", "last_name":"Parker"},{},{"id":4,"first_name":"John","last_name":"Doe"}]
```

Note that in both cases the same universal constructor `vector#1cb5c415` is used: in the request to serialize the value of type `Vector int`, and in the serialization of the value of type `Vector User` in the response. There is no ambiguity because in both cases the type of the value being (de)serialized is known before its (de)serialization begins. For example, after receiving the query, the server sees that the first part is `0x2d84d5f5`, which corresponds to the combinator `getUsers#2d84d5f5 (Vector int) = Vector User`. Thus, it is understood that what follows will be a value of type `Vector int`. After receiving the response to this query, the client knows that it must receive a value of type `Vector User` and it deserializes the response accordingly.

Current MTProto TL-schema

Below you will find the current MTProto TL-schema. More detais on TL »

See also the detailed schema in JSON »

```
int ? = Int;
long ? = Long;
double ? = Double;
string ? = String;

vector {t:Type} # [ t ] = Vector t;

int128 4*[ int ] = Int128;
int256 8*[ int ] = Int256;

resPQ#05162463 nonce:int128 server_nonce:int128 pq:bytes server_public_key_fingerprints:Vector<long> = ResPQ;

p_q_inner_data#83c95aec pq:bytes p:bytes q:bytes nonce:int128 server_nonce:int128 new_nonce:int256 = P_Q_inner_


server_DH_params_fail#79cb045d nonce:int128 server_nonce:int128 new_nonce_hash:int128 = Server_DH_Params;
server_DH_params_ok#d0e8075c nonce:int128 server_nonce:int128 encrypted_answer:bytes = Server_DH_Params;

server_DH_inner_data#b5890dba nonce:int128 server_nonce:int128 g:int dh_prime:bytes g_a:bytes server_time:int =

client_DH_inner_data#6643b654 nonce:int128 server_nonce:int128 retry_id:long g_b:bytes = Client_DH_Inner_Data;

dh_gen_ok#3bcbf734 nonce:int128 server_nonce:int128 new_nonce_hash1:int128 = Set_client_DH_params_answer;
dh_gen_retry#46dc1fb9 nonce:int128 server_nonce:int128 new_nonce_hash2:int128 = Set_client_DH_params_answer;
dh_gen_fail#a69dae02 nonce:int128 server_nonce:int128 new_nonce_hash3:int128 = Set_client_DH_params_answer;

rpc_result#f35c6d01 req_msg_id:long result:Object = RpcResult;
rpc_error#2144ca19 error_code:int error_message:string = RpcError;

rpc_answer_unknown#5e2ad36e = RpcDropAnswer;
rpc_answer_dropped_running#cd78e586 = RpcDropAnswer;
rpc_answer_dropped#a43ad8b7 msg_id:long seq_no:int bytes:int = RpcDropAnswer;

future_salt#0949d9dc valid_since:int valid_until:int salt:long = FutureSalt;
future_salts#ae500895 req_msg_id:long now:int salts:vector<future_salt> = FutureSalts;

pong#347773c5 msg_id:long ping_id:long = Pong;

destroy_session_ok#e22045fc session_id:long = DestroySessionRes;
destroy_session_none#62d350c9 session_id:long = DestroySessionRes;

new_session_created#9ec20908 first_msg_id:long unique_id:long server_salt:long = NewSession;

msg_container#73f1f8dc messages:vector<%Message> = MessageContainer;
message msg_id:long seqno:int bytes:int body:Object = Message;
msg_copy#e06046b2 orig_message:Message = MessageCopy;

gzip_packed#3072cfa1 packed_data:bytes = Object;

msgs_ack#62d6b459 msg_ids:Vector<long> = MsgsAck;

bad_msg_notification#a7eff811 bad_msg_id:long bad_msg_seqno:int error_code:int = BadMsgNotification;
bad_server_salt#edab447b bad_msg_id:long bad_msg_seqno:int error_code:int new_server_salt:long = BadMsgNotifica


msg_resend_req#7d861a08 msg_ids:Vector<long> = MsgResendReq;
msgs_state_req#da69fb52 msg_ids:Vector<long> = MsgsStateReq;
msgs_state_info#04deb57d req_msg_id:long info:bytes = MsgsStateInfo;
msgs_all_info#8cc0d131 msg_ids:Vector<long> info:bytes = MsgsAllInfo;
msg_detailed_info#276d3ec6 msg_id:long answer_msg_id:long bytes:int status:int = MsgDetailedInfo;
msg_new_detailed_info#809db6df answer_msg_id:long bytes:int status:int = MsgDetailedInfo;

---functions---

req_pq#60469778 nonce:int128 = ResPQ;

req_DH_params#d712e4be nonce:int128 server_nonce:int128 p:bytes q:bytes public_key_fingerprint:long encrypted_d

set_client_DH_params#f5045f1f nonce:int128 server_nonce:int128 encrypted_data:bytes = Set_client_DH_params_answ

rpc_drop_answer#58e4a740 req_msg_id:long = RpcDropAnswer;
get_future_salts#b921bd04 num:int = FutureSalts;
ping#7abe77ec ping_id:long = Pong;
ping_delay_disconnect#f3427b8c ping_id:long disconnect_delay:int = Pong;
destroy_session#e7512126 session_id:long = DestroySessionRes;

http_wait#9299359f max_delay:int wait_after:int max_wait:int = HttpWait;
```

End-to-End Encryption, Secret Chats

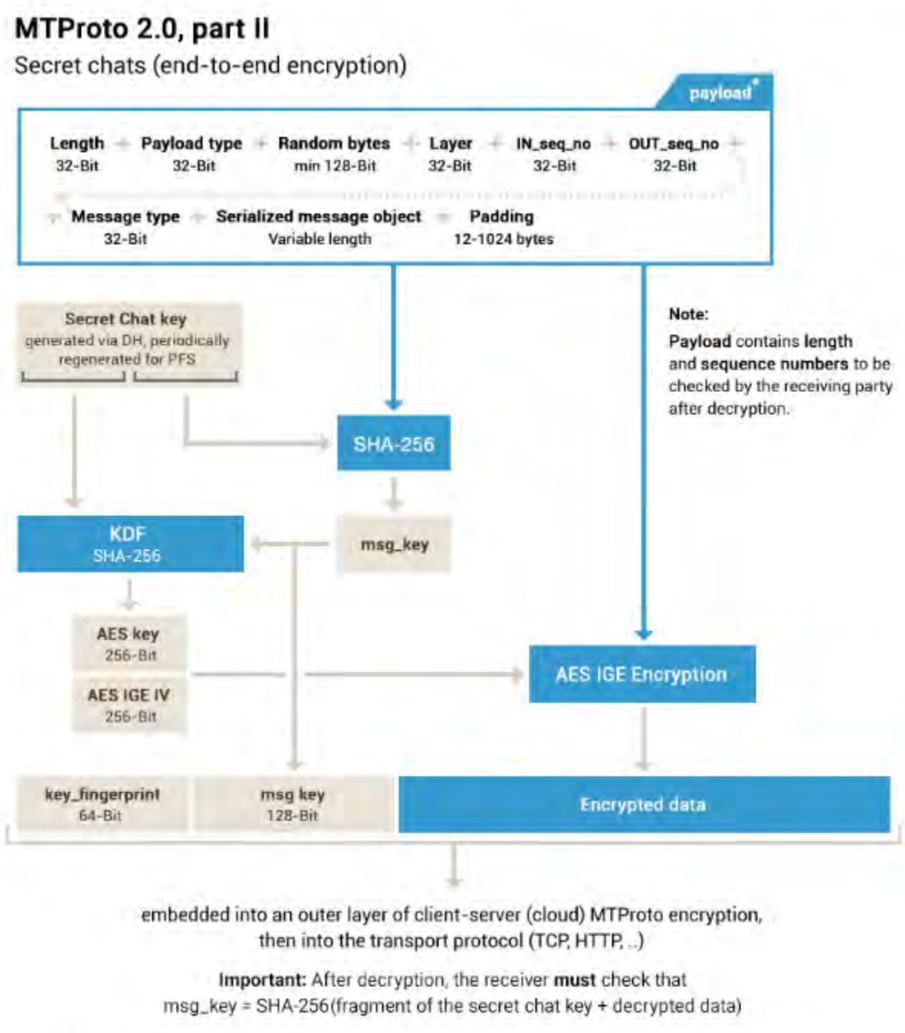
> This article on MTProto's End-to-End encryption is meant for **advanced users**. If you want to learn more about Secret Chats from a less intimidating source, kindly see our general FAQ.

> Note that as of version 4.6, major Telegram clients are using **MTProto 2.0**. MTProto v.1.0 is deprecated and is currently being phased out.

Related articles

Security guidelines for developers
Perfect Forward Secrecy in Secret Chats
Sequence numbers in Secret Chats
End-to-End TL Schema

Secret Chats are one-on-one chats wherein messages are encrypted with a key held only by the chat's participants. Note that the schema for these end-to-end encrypted Secret Chats is different from what is used for cloud chats:



A note on MTProto 2.0

This article describes the end-to-end encryption layer in the MTProto protocol version **2.0**. The principal differences from version 1.0 (described here for reference) are as follows:

- SHA-256 is used instead of SHA-1;
- Padding bytes are involved in the computation of msg_key;
- msg_key depends not only on the message to be encrypted, but on a portion of the secret chat key as well;
- 12..1024 padding bytes are used instead of 0..15 padding bytes in v.1.0.

See also: MTProto 2.0: Cloud Chats, server-client encryption

Key Generation

Keys are generated using the Diffie-Hellman protocol.

Let us consider the following scenario: User **A** would like to initiate end-to-end encrypted communication with User **B**.

Sending a Request

User **A** executes messages.getDhConfig to obtain the Diffie-Hellman parameters: a prime **p**, and a high order element **g**.

Executing this method before each new key generation procedure is of vital importance. It makes sense to cache the values of the parameters together with the version in order to avoid having to receive all of the values every time. If the version stored on the client is still up-to-date, the server will return the constructor messages.dhConfigNotModified.

Client is expected to check whether **p** is a safe 2048-bit prime (meaning that both **p** and **(p-1)/2** are prime, and that 2^2047 < p < 2^2048), and that **g** generates a cyclic subgroup of prime order **(p-1)/2**, i.e. is a quadratic residue **mod p**. Since **g** is always equal to 2, 3, 4, 5, 6 or 7, this is easily done using quadratic reciprocity law, yielding a simple condition on **p mod 4g** -- namely, **p mod 8 = 7** for **g = 2**; **p mod 3 = 2** for **g = 3**; no extra condition for **g = 4**; **p mod 5 = 1 or 4** for **g = 5**; **p mod 24 = 19 or 23** for **g = 6**; and **p mod 7 = 3, 5 or 6** for **g = 7**. After **g** and **p** have been checked by the client, it makes sense to cache the result, so as to avoid repeating lengthy computations in future. This cache might be shared with one used for Authorization Key generation.

If the client has an inadequate random number generator, it makes sense to pass the **random_length** parameter (random_length> 0) so the server generates its own random sequence **random** of the appropriate length. **Important**: using the server's random sequence in its raw form may be unsafe. It must be combined with a client sequence, for example, by generating a client random number of the same length (**client_random**) and using `final_random := random XOR client_random`.

Client **A** computes a 2048-bit number **a** (using sufficient entropy or the server's **random**; see above) and executes messages.requestEncryption after passing in `g_a := pow(g, a) mod dh_prime`.

User **B** receives the update updateEncryption for all associated authorization keys (all authorized devices) with the chat constructor encryptedChatRequested. The user must be shown basic information about User **A** and must be prompted to accept or reject the request.

Both clients are to check that **g**, **g_a** and **g_b** are greater than one and smaller than **p-1**. We recommend checking that **g_a** and **g_b** are between **2^{2048-64}** and **p - 2^{2048-64}** as well.

Accepting a Request

After User **B** confirms the creation of a secret chat with **A** in the client interface, Client **B** also receives up-to-date

configuration parameters for the Diffie-Hellman method. Thereafter, it generates a random 2048-bit number, **b**, using rules similar to those for **a**.

Having received **g_a** from the update with encryptedChatRequested, it can immediately generate the final shared key: `key = (pow(g_a, b) mod dh_prime)`. If key length < 256 bytes, add several leading zero bytes as padding — so that the key is exactly 256 bytes long. Its fingerprint, **key_fingerprint**, is equal to the 64 last bits of SHA1 (key).

Note 1: in this particular case SHA1 is used here even for MTProto 2.0 secret chats.

Note 2: this fingerprint is used as a sanity check for the key exchange procedure to detect bugs when developing client software — it is not connected to the key visualization used on the clients as means of external authentication in secret chats. Key visualizations on the clients are generated using the first 128 bits of SHA1(intial key) followed by the first 160 bits of SHA256(key used when secret chat was updated to layer 46).

Client **B** executes messages.acceptEncryption after passing it `g_b := pow(g, b) mod dh_prime` and **key_fingerprint**.

For all of Client **B's** authorized devices, except the current one, updateEncryption updates are sent with the constructor encryptedChatDiscarded. Thereafter, the only device that will be able to access the secret chat is Device **B**, which made the call to messages.acceptEncryption.

User **A** will be sent an updateEncryption update with the constructor encryptedChat, for the authorization key that initiated the chat.

With **g_b** from the update, Client **A** can also compute the shared key `key = (pow(g_b, a) mod dh_prime)`. If key length < 256 bytes, add several leading zero bytes as padding — so that the key is exactly 256 bytes long. If the fingerprint for the received key is identical to the one that was passed to encryptedChat, incoming messages can be sent and processed. Otherwise, messages.discardEncryption must be executed and the user notified.

Perfect Forward Secrecy

In order to keep past communications safe, official Telegram clients will initiate re-keying once a key has been used to decrypt and encrypt more than 100 messages, or has been in use for more than one week, provided the key has been used to encrypt at least one message. Old keys are then securely discarded and cannot be reconstructed, even with access to the new keys currently in use.

> The re-keying protocol is further described in this article: Perfect Forward Secrecy in Secret Chats.

Please note that your client must support Forward Secrecy in Secret Chats to be compatible with official Telegram clients.

Sending and Receiving Messages in a Secret Chat

Serialization and Encryption of Outgoing Messages

A TL object of type DecryptedMessage is created and contains the message in plain text. For backward compatibility, the object must be wrapped in the constructor decryptedMessageLayer with an indication of the supported layer (starting with 46).

> The TL-Schema for the contents of end-to-end encrypted messages is available here »

The resulting construct is serialized as an array of bytes using generic TL rules. The resulting array is prepended by 4 bytes containing the array length not counting these 4 bytes.

The byte array is padded with 12 to 1024 random padding bytes to make its length divisible by 16 bytes. (In the older MTProto 1.0 encryption, only 0 to 15 padding bytes were used.)

Message key, **msg_key**, is computed as the 128 middle bits of the SHA256 of the data obtained in the previous step, prepended by 32 bytes from the shared key **key**. (For the older MTProto 1.0 encryption, **msg_key** was computed differently, as the 128 lower bits of SHA1 of the data obtained in the previous steps, *excluding the padding bytes*.)

For MTProto 2.0, the AES key **aes_key** and initialization vector **aes_iv** are computed (**key** is the shared key obtained during Key Generation) as follows:

- msg_key_large = SHA256 (substr (key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (key, x, 36));
- sha256_b = SHA256 (substr (key, 40+x, 36) + msg_key);
- aes_key = substr (sha256_a, 0, 8) + substr (sha256_b, 8, 16) + substr (sha256_a, 24, 8);
- aes_iv = substr (sha256_b, 0, 8) + substr (sha256_a, 8, 16) + substr (sha256_b, 24, 8);

For MTProto 2.0, **x=0** for messages from the originator of the secret chat, **x=8** for the messages in the opposite direction.

For the obsolete MTProto 1.0, msg_key, aes_key, and aes_iv were computed differently (see this document for reference).

Data is encrypted with a 256-bit key, **aes_key**, and a 256-bit initialization vector, **aes-iv**, using AES-256 encryption with infinite garble extension (IGE). Encryption key fingerprint **key_fingerprint** and the message key **msg_key** are added at the top of the resulting byte array.

Encrypted data is embedded into a messages.sendEncrypted API call and passed to Telegram server for delivery to the other party of the Secret Chat.

Upgrading to MTProto 2.0 from MTProto 1.0

As soon as both parties in a secret chat are using at least Layer 73, they should only use MTProto 2.0 for all outgoing messages. Some of the first received messages may use MTProto 1.0, if a sufficiently high starting layer has not been negotiated during the creation of the secret chat. After the first message encrypted with MTProto 2.0 (or the first message with Layer 73 or higher) is received, all messages with higher sequence numbers must be encrypted with MTProto 2.0 as well.

As long as the current layer is lower than 73, each party should try to decrypt received messages with MTProto 1.0, and if this is not successfull (msg_key does not match), try MTProto 2.0. Once the first MTProto 2.0-encrypted message arrives (or the layer is upgraded to 73), there is no need to try MTProto 1.0 decryption for any of the further messages (unless the client is still waiting for some gaps to be closed).

Decrypting an Incoming Message

The steps above are performed in reverse order. When an encrypted message is received, you **must** check that msg_key is **in fact** equal to the 128 middle bits of the SHA256 hash of the decrypted message, prepended by 32 bytes taken from the shared **key**. If the message layer is greater than the one supported by the client, the user must be notified that the client version is out of date and prompted to update.

Sequence numbers

It is necessary to interpret all messages in their original order to protect against possible manipulations. Secret chats support a special mechanism for handling seq_no counters independently from the server.

> Proper handling of these counters is further described in this article: Sequence numbers in Secret Chats.

Please note that your client must support sequence numbers in Secret Chats to be compatible with official Telegram clients.

Sending Encrypted Files

All files sent to secret chats are encrypted with one-time keys that are in no way related to the chat's shared key. Before an encrypted file is sent, it is assumed that the encrypted file's address will be attached to the outside of an encrypted message using the **file** parameter of the messages.sendEncryptedFile method and that the key for direct decryption will be sent in the body of the message (the **key** parameter in the constructors decryptedMessageMediaPhoto, decryptedMessageMediaVideo and decryptedMessageMediaFile.

Prior to a file being sent to a secret chat, 2 random 256-bit numbers are computed which will serve as the AES key and initialization vector used to encrypt the file. AES-256 encryption with infinite garble extension (IGE) is used in like manner.

The key fingerprint is computed as follows:

- digest = md5(key + iv)
- fingerprint = substr(digest, 0, 4) XOR substr(digest, 4, 4)

The encrypted contents of a file are stored on the server in much the same way as those of a file in cloud chats: piece by piece using calls to upload.saveFilePart. A subsequent call to messages.sendEncryptedFile will assign an identifier to the stored file and send the address together with the message. The recipient will receive an update with encryptedMessage, and the **file** parameter will contain file information.

Incoming and outgoing encrypted files can be forwarded to other secret chats using the constructor inputEncryptedFile to avoid saving the same content on the server twice.

Working with an Update Box

Secret chats are associated with specific devices (or rather with authorization keys), not users. A conventional message box, which uses **pts** to describe the client's status, is not suitable, because it is designed for long-term message storage and message access from different devices.

An additional temporary message queue is introduced as a solution to this problem. When an update regarding a message from a secret chat is sent, a new value of **qts** is sent, which helps reconstruct the difference if there has been a long break in the connection or in case of loss of an update.

As the number of events increases, the value of **qts** increases by 1 with each new event. The initial value may not (and will not) be equal to 0.

The fact that events from the temporary queue have been received and stored by the client is acknowledged explicitly by a call to the messages.receivedQueue method or implicitly by a call to updates.getDifference (the value of **qts** passed, not the final state). All messages acknowledged as delivered by the client, as well as any messages older than 7 days, may (and will) be deleted from the server.

Upon de-authorization, the event queue of the corresponding device will be forcibly cleared, and the value of **qts** will become irrelevant.

Updating to new layers

Your client should always store the maximal layer that is known to be supported by the client on the other side of a secret chat. When the secret chat is first created, this value should be initialized to 46. This remote layer value must always be updated immediately after receiving *any* packet containing information of an upper layer, i.e.:

- any secret chat message containing *layer_no* in its `decryptedMessageLayer` with *layer*>=46, or
- a decryptedMessageActionNotifyLayer service message, wrapped as if it were the decryptedMessageService constructor of the obsolete layer 8 (constructor `decryptedMessageService#aa48327d`).

Notifying the remote client about your local layer

In order to notify the remote client of your local layer, your client must send a message of the `decryptedMessageActionNotifyLayer` type. This notification must be wrapped in a constructor of an appropriate layer.

There are two cases when your client must notify the remote client about its local layer:

1. As soon as a new secret chat has been created, immediately after the secret key has been successfully exchanged.
2. Immediately after the local client has been updated to support a new secret chat layer. In this case notifications must be sent to **all** currently existing secret chats. Note that this is only necessary when updating to new layers that contain changes in the secret chats implementation (e.g. you don't need to do this when your client is updated from Layer 46 to Layer 47).

Current end-to-end TL-schema

Below you will find the current end-to-end TL-schema. More details on TL »

See also:
End-to-end encryption in MTProto, Secret Chats
Detailed schema in JSON

Layer 105

```
===8===
decryptedMessage#1f814f1f random_id:long random_bytes:bytes message:string media:DecryptedMessageMedia = Decryp
decryptedMessageService#aa48327d random_id:long random_bytes:bytes action:DecryptedMessageAction = DecryptedMes
decryptedMessageMediaEmpty#89f5c4a = DecryptedMessageMedia;
decryptedMessageMediaPhoto#32798a8c thumb:bytes thumb_w:int thumb_h:int w:int h:int size:int key:bytes iv:bytes
decryptedMessageMediaVideo#4cee6ef3 thumb:bytes thumb_w:int thumb_h:int duration:int w:int h:int size:int key:b
decryptedMessageMediaGeoPoint#35480a59 lat:double long:double = DecryptedMessageMedia;
decryptedMessageMediaContact#588a0a97 phone_number:string first_name:string last_name:string user_id:int = Decr
decryptedMessageActionSetMessageTTL#a1733aec ttl_seconds:int = DecryptedMessageAction;
decryptedMessageMediaDocument#b095434b thumb:bytes thumb_w:int thumb_h:int file_name:string mime_type:string si
decryptedMessageMediaAudio#6080758f duration:int size:int key:bytes iv:bytes = DecryptedMessageMedia;
decryptedMessageActionReadMessages#c4f40be random_ids:Vector<long> = DecryptedMessageAction;
decryptedMessageActionDeleteMessages#65614304 random_ids:Vector<long> = DecryptedMessageAction;
decryptedMessageActionScreenshotMessages#8ac1f475 random_ids:Vector<long> = DecryptedMessageAction;
decryptedMessageActionFlushHistory#6719e45c = DecryptedMessageAction;

===17===
decryptedMessage#204d3878 random_id:long ttl:int message:string media:DecryptedMessageMedia = DecryptedMessage;
decryptedMessageService#73164160 random_id:long action:DecryptedMessageAction = DecryptedMessage;
decryptedMessageMediaVideo#524a415d thumb:bytes thumb_w:int thumb_h:int duration:int mime_type:string w:int h:i
decryptedMessageMediaAudio#57e0a9cb duration:int mime_type:string size:int key:bytes iv:bytes = DecryptedMessag
decryptedMessageLayer#1be31789 random_bytes:bytes layer:int in_seq_no:int out_seq_no:int message:DecryptedMessa
sendMessageTypingAction#16bf744e = SendMessageAction;
sendMessageCancelAction#fd5ec8f5 = SendMessageAction;
sendMessageRecordVideoAction#a187d66f = SendMessageAction;
sendMessageUploadVideoAction#92042ff7 = SendMessageAction;
sendMessageRecordAudioAction#d52f73f7 = SendMessageAction;
sendMessageUploadAudioAction#e6ac8a6f = SendMessageAction;
sendMessageUploadPhotoAction#990a3c1a = SendMessageAction;
sendMessageUploadDocumentAction#8faee98e = SendMessageAction;
sendMessageGeoLocationAction#176f8ba1 = SendMessageAction;
sendMessageChooseContactAction#628cbc6f = SendMessageAction;
decryptedMessageActionResend#511110b0 start_seq_no:int end_seq_no:int = DecryptedMessageAction;
decryptedMessageActionNotifyLayer#f3048883 layer:int = DecryptedMessageAction;
decryptedMessageActionTyping#ccb27641 action:SendMessageAction = DecryptedMessageAction;

===20===
decryptedMessageActionRequestKey#f3c9611b exchange_id:long g_a:bytes = DecryptedMessageAction;
decryptedMessageActionAcceptKey#6fe1735b exchange_id:long g_b:bytes key_fingerprint:long = DecryptedMessageActi
decryptedMessageActionAbortKey#dd05ec6b exchange_id:long = DecryptedMessageAction;
decryptedMessageActionCommitKey#ec2e0b9b exchange_id:long key_fingerprint:long = DecryptedMessageAction;
decryptedMessageActionNoop#a82fdd63 = DecryptedMessageAction;

===23===
documentAttributeImageSize#6c37c15c w:int h:int = DocumentAttribute;
documentAttributeAnimated#11b58939 = DocumentAttribute;
documentAttributeSticker#fb0a5727 = DocumentAttribute;
documentAttributeVideo#5910cccb duration:int w:int h:int = DocumentAttribute;
documentAttributeAudio#51448e5 duration:int = DocumentAttribute;
documentAttributeFilename#15590068 file_name:string = DocumentAttribute;
photoSizeEmpty#e17e23c type:string = PhotoSize;
photoSize#77bfb61b type:string location:FileLocation w:int h:int size:int = PhotoSize;
photoCachedSize#e9a734fa type:string location:FileLocation w:int h:int bytes:bytes = PhotoSize;
fileLocationUnavailable#7c596b46 volume_id:long local_id:int secret:long = FileLocation;
fileLocation#53d69076 dc_id:int volume_id:long local_id:int secret:long = FileLocation;
decryptedMessageMediaExternalDocument#fa95b0dd id:long access_hash:long date:int mime_type:string size:int thum

===45===
decryptedMessage#36b091de flags:# random_id:long ttl:int message:string media:flags.9?DecryptedMessageMedia ent
decryptedMessageMediaPhoto#f1fa8d78 thumb:bytes thumb_w:int thumb_h:int w:int h:int size:int key:bytes iv:bytes
decryptedMessageMediaVideo#970c8c0e thumb:bytes thumb_w:int thumb_h:int duration:int mime_type:string w:int h:i
decryptedMessageMediaDocument#7afe8ae2 thumb:bytes thumb_w:int thumb_h:int mime_type:string size:int key:bytes
documentAttributeSticker#3a556302 alt:string stickerset:InputStickerSet = DocumentAttribute;
documentAttributeAudio#ded218e0 duration:int title:string performer:string = DocumentAttribute;
messageEntityUnknown#bb92ba95 offset:int length:int = MessageEntity;
messageEntityMention#fa04579d offset:int length:int = MessageEntity;
messageEntityHashtag#6f635b0d offset:int length:int = MessageEntity;
messageEntityBotCommand#6cef8ac7 offset:int length:int = MessageEntity;
messageEntityUrl#6ed02538 offset:int length:int = MessageEntity;
messageEntityEmail#64e475c2 offset:int length:int = MessageEntity;
messageEntityBold#bd610bc9 offset:int length:int = MessageEntity;
messageEntityItalic#826f8b60 offset:int length:int = MessageEntity;
messageEntityCode#28a20571 offset:int length:int = MessageEntity;
messageEntityPre#73924be0 offset:int length:int language:string = MessageEntity;
messageEntityTextUrl#76a6d327 offset:int length:int url:string = MessageEntity;
inputStickerSetShortName#861cc8a0 short_name:string = InputStickerSet;
inputStickerSetEmpty#ffb62b95 = InputStickerSet;
decryptedMessageMediaVenue#8a0df56f lat:double long:double title:string address:string provider:string venue_id
decryptedMessageMediaWebPage#e50511d8 url:string = DecryptedMessageMedia;

===46===
documentAttributeAudio#9852f9c6 flags:# voice:flags.10?true duration:int title:flags.0?string performer:flags.1

===66===
documentAttributeVideo#ef02ce6 flags:# round_message:flags.0?true duration:int w:int h:int = DocumentAttribute;
sendMessageRecordRoundAction#88f27fbc = SendMessageAction;
sendMessageUploadRoundAction#bb718624 = SendMessageAction;

===73===
decryptedMessage#91cc4674 flags:# random_id:long ttl:int message:string media:flags.9?DecryptedMessageMedia ent
```

Security Guidelines for Client Developers

See also:

Perfect Forward Secrecy
Secret chats, end-to-end encryption
Perfect Forward Secrecy in Secret Chats
MTProto 2.0, Detailed Description

While MTProto is designed to be a reasonably fast and secure protocol, its advantages can be easily negated by careless implementation. We collected some security guidelines for client software developers on this page. All Telegram clients are required to comply.

> Note that as of version 4.6, major Telegram clients are using **MTProto 2.0**.
> MTProto v.1.0 is deprecated and is currently being phased out.

Diffie—Hellman key exchange

We use DH key exchange in two cases:

- Creating an authorization key
- Establishing Secret Chats with end-to-end encryption

In both cases, there are some verifications to be done whenever DH is used:

Validation of DH parameters

Client is expected to check whether **p** = **dh_prime** is a safe 2048-bit prime (meaning that both **p** and **(p-1)/2** are prime, and that $2^{2047} < p < 2^{2048}$), and that **g** generates a cyclic subgroup of prime order **(p-1)/2**, i.e. is a quadratic residue **mod p**. Since **g** is always equal to 2, 3, 4, 5, 6 or 7, this is easily done using quadratic reciprocity law, yielding a simple condition on **p mod 4g** — namely, **p mod 8 = 7** for **g = 2**; **p mod 3 = 2** for **g = 3**; no extra condition for **g = 4**; **p mod 5 = 1 or 4** for **g = 5**; **p mod 24 = 19 or 23** for **g = 6**; and **p mod 7 = 3, 5 or 6** for **g = 7**. After **g** and **p** have been checked by the client, it makes sense to cache the result, so as not to repeat lengthy computations in future.

If the verification takes too long (which is the case for older mobile devices), one might initially run only 15 Miller—Rabin iterations (use parameter 30 in Java) for verifying primeness of **p** and **(p - 1)/2** with error probability not exceeding one billionth, and do more iterations in the background later.

Another way to optimize this is to embed into the client application code a small table with some known "good" couples **(g,p)** (or just known safe primes **p**, since the condition on **g** is easily verified during execution), checked during code generation phase, so as to avoid doing such verification during runtime altogether. The server rarely changes these values, thus one usually needs to put the current value of server's **dh_prime** into such a table. For example, the current value of **dh_prime** equals (in big-endian byte order)

```
C7 1C AE B9 C6 B1 C9 04 8E 6C 52 2F 70 F1 3F 73 98 0D 40 23 8E 3E 21 C1 49 34 D0 37 56 3D 93 0F 48 19 8A 0A A7
```

g_a and g_b validation

Apart from the conditions on the Diffie-Hellman prime **dh_prime** and generator **g**, both sides are to check that **g**, **g_a** and **g_b** are greater than **1** and less than **dh_prime - 1**. We recommend checking that **g_a** and **g_b** are between **2^{2048-64}** and **dh_prime - 2^{2048-64}** as well.

Checking SHA1 hash values during key generation

Once the client receives a `server_DH_params_ok` answer in step 5) of the Authorization Key generation protocol and decrypts it obtaining `answer_with_hash`, it MUST check that

```
answer_with_hash := SHA1(answer) + answer + (0-15 random bytes)
```

In other words, the first 20 bytes of `answer_with_hash` must be equal to SHA1 of the remainder of the decrypted message without the padding random bytes.

Checking nonce, server_nonce and new_nonce fields

When the client receives and/or decrypts server messages during creation of Authorization Key, and these messages contain some nonce fields already known to the client from messages previously obtained during the same run of the protocol, the client is to check that these fields indeed contain the values previosly known.

Using secure pseudorandom number generator to create DH secret parameters `a` and `b`

Client must use a cryptographically secure PRNG to generate secret exponents `a` or `b` for DH key exchange. For secret chats, the client might request some entropy (random bytes) from the server while invoking messages.getDhConfig and feed these random bytes into its PRNG (for example, by `PRNG_seed` if OpenSSL library is used), but never using these "random" bytes by themselves or replacing by them the local PRNG seed. One should mix bytes received from server into local PRNG seed.

MTProto Encrypted Messages

Some important checks are to be done while sending and especially receiving encrypted MTProto messages.

Checking SHA256 hash value of msg_key

`msg_key` is used not only to compute the AES key and IV to decrypt the received message. After decryption, the client **MUST** check that `msg_key` is indeed equal to SHA256 of the plaintext obtained as the result of decryption (including the final 12...1024 padding bytes), prepended with 32 bytes taken from the `auth_key`, as explained in MTProto 2.0 Description.

If an error is encountered before this check could be performed, the client **must** perform the `msg_key` check anyway before returning any result. Note that the response to any error encountered before the `msg_key` check **must** be the same as the response to a failed `msg_key` check.

Checking message length

The client **must** check that the length of the message or container obtained from the decrypted message (computed from its `length` field) does not exceed the total size of the plaintext, and that the difference (i.e. the length of the random padding) lies in the range from 12 to 1024 bytes.

The length should be always divisible by 4 and non-negative. On no account the client is to access data past the end of the decryption buffer containing the plaintext message.

Checking session_id

The client is to check that the `session_id` field in the decrypted message indeed equals to that of an active session created by the client.

Checking msg_id

The client must check that `msg_id` has even parity for messages from client to server, and odd parity for messages from server to client.

In addition, the identifiers (msg_id) of the last N messages received from the other side must be stored, and if a message comes in with an msg_id lower than all or equal to any of the stored values, that message is to be ignored. Otherwise, the new message msg_id is added to the set, and, if the number of stored msg_id values is greater than N, the oldest (i. e. the lowest) is discarded.

In addition, msg_id values that belong over 30 seconds in the future or over 300 seconds in the past are to be ignored (recall that `msg_id` approximately equals unixtime * 2^32). This is especially important for the server. The client would also find this useful (to protect from a replay attack), but only if it is certain of its time (for example, if its time has been synchronized with that of the server).

Certain client-to-server service messages containing data sent by the client to the server (for example, `msg_id` of a recent client query) may, nonetheless, be processed on the client even if the time appears to be "incorrect". This is especially true of messages to change server_salt and notifications about invalid time on the client. See Mobile Protocol: Service Messages.

Behavior in case of mismatch

If one of the checks listed above fails, the client is to completely discard the message obtained from server. We also recommend closing and reestablishing the TCP connection to the server, then retrying the operation or the whole key generation protocol.

No information from incorrect messages can be used. Even if the application throws an exception and dies, this is much better than continuing with invalid data.

Notice that invalid messages will infrequently appear during normal work even if no malicious tampering is being done. This is due to network transmission errors. We recommend ignoring the invalid message and closing the TCP connection, then creating a new TCP connection to the server and retrying the original query.

> The previous version of security recommendations relevant for MTProto 1.0 clients is available here.

End-to-End Encryption, Secret Chats

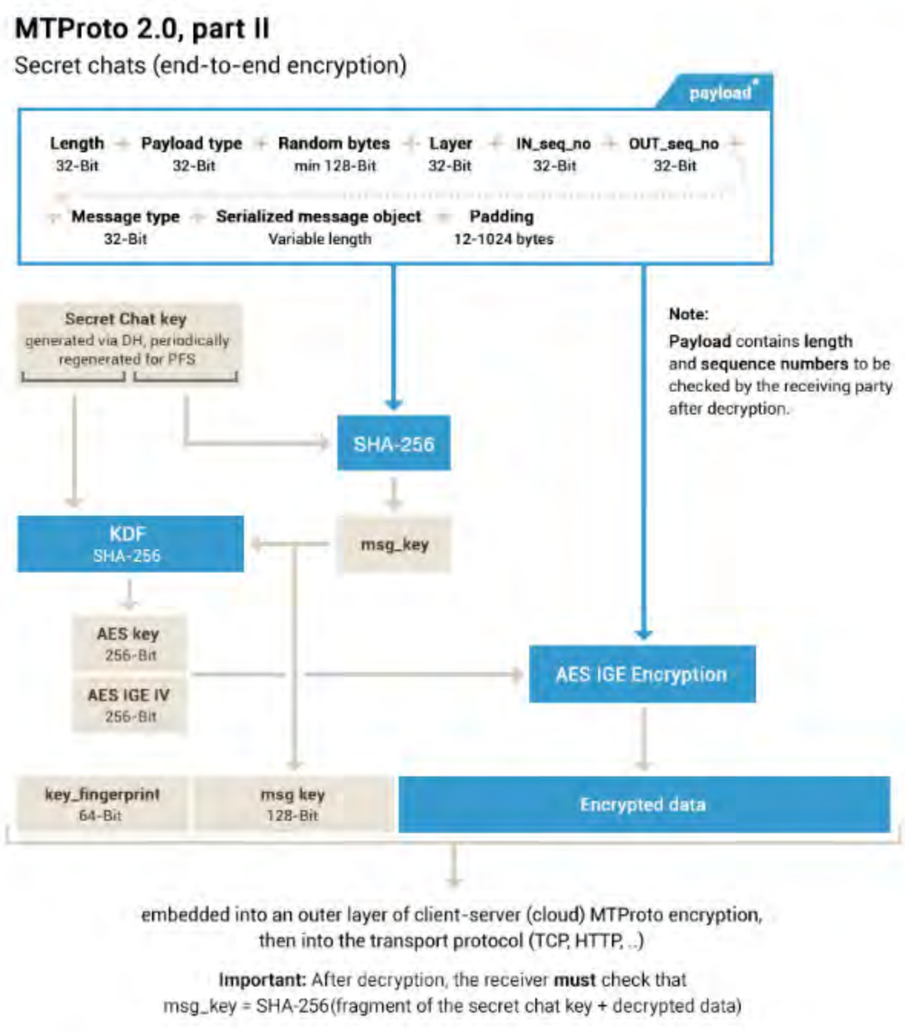
> This article on MTProto's End-to-End encryption is meant for **advanced users**. If you want to learn more about Secret Chats from a less intimidating source, kindly see our general FAQ.

> Note that as of version 4.6, major Telegram clients are using **MTProto 2.0**. MTProto v.1.0 is deprecated and is currently being phased out.

Related articles

Security guidelines for developers
Perfect Forward Secrecy in Secret Chats
Sequence numbers in Secret Chats
End-to-End TL Schema

Secret Chats are one-on-one chats wherein messages are encrypted with a key held only by the chat's participants. Note that the schema for these end-to-end encrypted Secret Chats is different from what is used for cloud chats:



A note on MTProto 2.0

This article describes the end-to-end encryption layer in the MTProto protocol version **2.0**. The principal differences from version 1.0 (described here for reference) are as follows:

- SHA-256 is used instead of SHA-1;
- Padding bytes are involved in the computation of msg_key;
- msg_key depends not only on the message to be encrypted, but on a portion of the secret chat key as well;
- 12..1024 padding bytes are used instead of 0..15 padding bytes in v.1.0.

See also: MTProto 2.0: Cloud Chats, server-client encryption

Key Generation

Keys are generated using the Diffie-Hellman protocol.

Let us consider the following scenario: User **A** would like to initiate end-to-end encrypted communication with User **B**.

Sending a Request

User **A** executes messages.getDhConfig to obtain the Diffie-Hellman parameters: a prime **p**, and a high order element **g**.

Executing this method before each new key generation procedure is of vital importance. It makes sense to cache the values of the parameters together with the version in order to avoid having to receive all of the values every time. If the version stored on the client is still up-to-date, the server will return the constructor messages.dhConfigNotModified.

Client is expected to check whether **p** is a safe 2048-bit prime (meaning that both **p** and **(p-1)/2** are prime, and that 2^2047 < p < 2^2048), and that **g** generates a cyclic subgroup of prime order **(p-1)/2**, i.e. is a quadratic residue **mod p**. Since **g** is always equal to 2, 3, 4, 5, 6 or 7, this is easily done using quadratic reciprocity law, yielding a simple condition on **p mod 4g** -- namely, **p mod 8 = 7** for **g = 2**; **p mod 3 = 2** for **g = 3**; no extra condition for **g = 4**; **p mod 5 = 1 or 4** for **g = 5**; **p mod 24 = 19 or 23** for **g = 6**; and **p mod 7 = 3, 5 or 6** for **g = 7**. After **g** and **p** have been checked by the client, it makes sense to cache the result, so as to avoid repeating lengthy computations in future. This cache might be shared with one used for Authorization Key generation.

If the client has an inadequate random number generator, it makes sense to pass the **random_length** parameter (random_length> 0) so the server generates its own random sequence **random** of the appropriate length. **Important**: using the server's random sequence in its raw form may be unsafe. It must be combined with a client sequence, for example, by generating a client random number of the same length (**client_random**) and using `final_random := random XOR client_random`.

Client **A** computes a 2048-bit number **a** (using sufficient entropy or the server's **random**; see above) and executes messages.requestEncryption after passing in `g_a := pow(g, a) mod dh_prime`.

User **B** receives the update updateEncryption for all associated authorization keys (all authorized devices) with the chat constructor encryptedChatRequested. The user must be shown basic information about User **A** and must be prompted to accept or reject the request.

Both clients are to check that **g**, **g_a** and **g_b** are greater than one and smaller than **p-1**. We recommend checking that **g_a** and **g_b** are between **2^{2048-64}** and **p - 2^{2048-64}** as well.

Accepting a Request

After User **B** confirms the creation of a secret chat with **A** in the client interface, Client **B** also receives up-to-date

configuration parameters for the Diffie-Hellman method. Thereafter, it generates a random 2048-bit number, **b**, using rules similar to those for **a**.

Having received **g_a** from the update with encryptedChatRequested, it can immediately generate the final shared key: `key = (pow(g_a, b) mod dh_prime)`. If key length < 256 bytes, add several leading zero bytes as padding — so that the key is exactly 256 bytes long. Its fingerprint, **key_fingerprint**, is equal to the 64 last bits of SHA1 (key).

Note 1: in this particular case SHA1 is used here even for MTProto 2.0 secret chats.

Note 2: this fingerprint is used as a sanity check for the key exchange procedure to detect bugs when developing client software — it is not connected to the key visualization used on the clients as means of external authentication in secret chats. Key visualizations on the clients are generated using the first 128 bits of SHA1(intial key) followed by the first 160 bits of SHA256(key used when secret chat was updated to layer 46).

Client **B** executes messages.acceptEncryption after passing it `g_b := pow(g, b) mod dh_prime` and **key_fingerprint**.

For all of Client **B's** authorized devices, except the current one, updateEncryption updates are sent with the constructor encryptedChatDiscarded. Thereafter, the only device that will be able to access the secret chat is Device **B**, which made the call to messages.acceptEncryption.

User **A** will be sent an updateEncryption update with the constructor encryptedChat, for the authorization key that initiated the chat.

With **g_b** from the update, Client **A** can also compute the shared key `key = (pow(g_b, a) mod dh_prime)`. If key length < 256 bytes, add several leading zero bytes as padding — so that the key is exactly 256 bytes long. If the fingerprint for the received key is identical to the one that was passed to encryptedChat, incoming messages can be sent and processed. Otherwise, messages.discardEncryption must be executed and the user notified.

Perfect Forward Secrecy

In order to keep past communications safe, official Telegram clients will initiate re-keying once a key has been used to decrypt and encrypt more than 100 messages, or has been in use for more than one week, provided the key has been used to encrypt at least one message. Old keys are then securely discarded and cannot be reconstructed, even with access to the new keys currently in use.

> The re-keying protocol is further described in this article: Perfect Forward Secrecy in Secret Chats.

Please note that your client must support Forward Secrecy in Secret Chats to be compatible with official Telegram clients.

Sending and Receiving Messages in a Secret Chat

Serialization and Encryption of Outgoing Messages

A TL object of type DecryptedMessage is created and contains the message in plain text. For backward compatibility, the object must be wrapped in the constructor decryptedMessageLayer with an indication of the supported layer (starting with 46).

> The TL-Schema for the contents of end-to-end encrypted messages is available here »

The resulting construct is serialized as an array of bytes using generic TL rules. The resulting array is prepended by 4 bytes containing the array length not counting these 4 bytes.

The byte array is padded with 12 to 1024 random padding bytes to make its length divisible by 16 bytes. (In the older MTProto 1.0 encryption, only 0 to 15 padding bytes were used.)

Message key, **msg_key**, is computed as the 128 middle bits of the SHA256 of the data obtained in the previous step, prepended by 32 bytes from the shared key **key**. (For the older MTProto 1.0 encryption, **msg_key** was computed differently, as the 128 lower bits of SHA1 of the data obtained in the previous steps, *excluding the padding bytes*.)

For MTProto 2.0, the AES key **aes_key** and initialization vector **aes_iv** are computed (**key** is the shared key obtained during Key Generation) as follows:

- msg_key_large = SHA256 (substr (key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (key, x, 36));
- sha256_b = SHA256 (substr (key, 40+x, 36) + msg_key);
- aes_key = substr (sha256_a, 0, 8) + substr (sha256_b, 8, 16) + substr (sha256_a, 24, 8);
- aes_iv = substr (sha256_b, 0, 8) + substr (sha256_a, 8, 16) + substr (sha256_b, 24, 8);

For MTProto 2.0, **x=0** for messages from the originator of the secret chat, **x=8** for the messages in the opposite direction.

For the obsolete MTProto 1.0, msg_key, aes_key, and aes_iv were computed differently (see this document for reference).

Data is encrypted with a 256-bit key, **aes_key**, and a 256-bit initialization vector, **aes-iv**, using AES-256 encryption with infinite garble extension (IGE). Encryption key fingerprint **key_fingerprint** and the message key **msg_key** are added at the top of the resulting byte array.

Encrypted data is embedded into a messages.sendEncrypted API call and passed to Telegram server for delivery to the other party of the Secret Chat.

Upgrading to MTProto 2.0 from MTProto 1.0

As soon as both parties in a secret chat are using at least Layer 73, they should only use MTProto 2.0 for all outgoing messages. Some of the first received messages may use MTProto 1.0, if a sufficiently high starting layer has not been negotiated during the creation of the secret chat. After the first message encrypted with MTProto 2.0 (or the first message with Layer 73 or higher) is received, all messages with higher sequence numbers must be encrypted with MTProto 2.0 as well.

As long as the current layer is lower than 73, each party should try to decrypt received messages with MTProto 1.0, and if this is not successfull (msg_key does not match), try MTProto 2.0. Once the first MTProto 2.0-encrypted message arrives (or the layer is upgraded to 73), there is no need to try MTProto 1.0 decryption for any of the further messages (unless the client is still waiting for some gaps to be closed).

Decrypting an Incoming Message

The steps above are performed in reverse order. When an encrypted message is received, you **must** check that msg_key is **in fact** equal to the 128 middle bits of the SHA256 hash of the decrypted message, prepended by 32 bytes taken from the shared **key**. If the message layer is greater than the one supported by the client, the user must be notified that the client version is out of date and prompted to update.

Sequence numbers

It is necessary to interpret all messages in their original order to protect against possible manipulations. Secret chats support a special mechanism for handling seq_no counters independently from the server.

> Proper handling of these counters is further described in this article: Sequence numbers in Secret Chats.

Please note that your client must support sequence numbers in Secret Chats to be compatible with official Telegram clients.

Sending Encrypted Files

All files sent to secret chats are encrypted with one-time keys that are in no way related to the chat's shared key. Before an encrypted file is sent, it is assumed that the encrypted file's address will be attached to the outside of an encrypted message using the **file** parameter of the messages.sendEncryptedFile method and that the key for direct decryption will be sent in the body of the message (the **key** parameter in the constructors decryptedMessageMediaPhoto, decryptedMessageMediaVideo and decryptedMessageMediaFile.

Prior to a file being sent to a secret chat, 2 random 256-bit numbers are computed which will serve as the AES key and initialization vector used to encrypt the file. AES-256 encryption with infinite garble extension (IGE) is used in like manner.

The key fingerprint is computed as follows:

- digest = md5(key + iv)
- fingerprint = substr(digest, 0, 4) XOR substr(digest, 4, 4)

The encrypted contents of a file are stored on the server in much the same way as those of a file in cloud chats: piece by piece using calls to upload.saveFilePart. A subsequent call to messages.sendEncryptedFile will assign an identifier to the stored file and send the address together with the message. The recipient will receive an update with encryptedMessage, and the **file** parameter will contain file information.

Incoming and outgoing encrypted files can be forwarded to other secret chats using the constructor inputEncryptedFile to avoid saving the same content on the server twice.

Working with an Update Box

Secret chats are associated with specific devices (or rather with authorization keys), not users. A conventional message box, which uses **pts** to describe the client's status, is not suitable, because it is designed for long-term message storage and message access from different devices.

An additional temporary message queue is introduced as a solution to this problem. When an update regarding a message from a secret chat is sent, a new value of **qts** is sent, which helps reconstruct the difference if there has been a long break in the connection or in case of loss of an update.

As the number of events increases, the value of **qts** increases by 1 with each new event. The initial value may not (and will not) be equal to 0.

The fact that events from the temporary queue have been received and stored by the client is acknowledged explicitly by a call to the messages.receivedQueue method or implicitly by a call to updates.getDifference (the value of **qts** passed, not the final state). All messages acknowledged as delivered by the client, as well as any messages older than 7 days, may (and will) be deleted from the server.

Upon de-authorization, the event queue of the corresponding device will be forcibly cleared, and the value of **qts** will become irrelevant.

Updating to new layers

Your client should always store the maximal layer that is known to be supported by the client on the other side of a secret chat. When the secret chat is first created, this value should be initialized to 46. This remote layer value must always be updated immediately after receiving *any* packet containing information of an upper layer, i.e.:

- any secret chat message containing *layer_no* in its `decryptedMessageLayer` with *layer*>=46, or
- a decryptedMessageActionNotifyLayer service message, wrapped as if it were the decryptedMessageService constructor of the obsolete layer 8 (constructor `decryptedMessageService#aa48327d`).

Notifying the remote client about your local layer

In order to notify the remote client of your local layer, your client must send a message of the `decryptedMessageActionNotifyLayer` type. This notification must be wrapped in a constructor of an appropriate layer.

There are two cases when your client must notify the remote client about its local layer:

1. As soon as a new secret chat has been created, immediately after the secret key has been successfully exchanged.
2. Immediately after the local client has been updated to support a new secret chat layer. In this case notifications must be sent to **all** currently existing secret chats. Note that this is only necessary when updating to new layers that contain changes in the secret chats implementation (e.g. you don't need to do this when your client is updated from Layer 46 to Layer 47).

End-to-End Encrypted Voice Calls

This article describes the end-to-end encryption used for Telegram voice calls.

Related articles

End-to-End Encryption in Secret Chats
Security Guidelines for Client Developers

Establishing voice calls

Before a voice call is ready, some preliminary actions have to be performed. The calling party needs to contact the party to be called and check whether it is ready to accept the call. Besides that, the parties have to negotiate the protocols to be used, learn the IP addresses of each other or of the Telegram relay servers to be used (so-called *reflectors*), and generate a one-time encryption key for this voice call with the aid of *Diffie—Hellman key exchange*. All of this is accomplished in parallel with the aid of several Telegram API methods and related notifications. This document details the generation of the encryption key. Other negotiations will be eventually documented elsewhere.

Key Generation

The Diffie-Hellman key exchange, as well as the whole protocol used to create a new voice call, is quite similar to the one used for Secret Chats. We recommend studying the linked article before proceeding.

However, we have introduced some important changes to facilitate the key verification process. Below is the entire exchange between the two communicating parties, the Caller (A) and the Callee (B), through the Telegram servers (S).

- *A* executes messages.getDhConfig to find out the 2048-bit Diffie-Hellman prime *p* and generator *g*. The client is expected to check whether *p* is a safe prime and perform all the security checks necessary for secret chats.
- *A* chooses a random value of *a*, 1 < a < p-1, and computes *g_a:=power(g,a) mod p* (a 256-byte number) and *g_a_hash:=SHA256(g_a)* (32 bytes long).
- *A* invokes (sends to server *S*) phone.requestCall, which has the field `g_a_hash:bytes`, among others. For this call, this field is to be filled with *g_a_hash*, **not** *g_a* itself.
- The Server *S* performs privacy checks and sends an updatePhoneCall update with a phoneCallRequested constructor to all of *B*'s active devices. This update, apart from the identity of *A* and other relevant parameters, contains the *g_a_hash* field, filled with the value obtained from *A*.
- *B* accepts the call on one of their devices, stores the received value of *g_a_hash* for this instance of the voice call creation protocol, chooses a random value of *b*, 1 < b < p-1, computes *g_b:=power(g,b) mod p*, performs all the required security checks, and invokes the phone.acceptCall method, which has a *g_b:bytes* field (among others), to be filled with the value of *g_b* itself (not its hash).
- The Server *S* sends an updatePhoneCall with the phoneCallDiscarded constructor to all other devices *B* has authorized, to prevent accepting the same call on any of the other devices. From this point on, the server *S* works only with that of *B*'s devices which has invoked phone.acceptCall first.
- The Server *S* sends to *A* an updatePhoneCall update with phoneCallAccepted constructor, containing the value of *g_b* received from *B*.
- *A* performs all the usual security checks on *g_b* and *a*, computes the Diffie—Hellman key *key:=power(g_b,a) mod p* and its fingerprint *key_fingerprint:long*, equal to the lower 64 bits of *SHA1(key)*, the same as with secret chats. Then *A* invokes the phone.confirmCall method, containing `g_a:bytes` and `key_fingerprint:long`.
- The Server *S* sends to *B* an updatePhoneCall update with the phoneCall constructor, containing the value of *g_a* in *g_a_or_b:bytes* field, and *key_fingerprint:long*
- At this point *B* receives the value of *g_a*. It checks that *SHA256(g_a)* is indeed equal to the previously received value of *g_a_hash*, performs all the usual Diffie-Hellman security checks, and computes the key *key:=power(g_a,b) mod p* and its fingerprint, equal to the lower 64 bits of *SHA1(key)*. Then it checks that this fingerprint equals the value of `key_fingerprint:long` received from the other side, as an implementation sanity check.

At this point, the Diffie—Hellman key exchange is complete, and both parties have a 256-byte shared secret key *key* which is used to encrypt all further exchanges between *A* and *B*.

It is of paramount importance to accept each update only once for each instance of the key generation protocol, discarding any duplicates or alternative versions of already received and processed messages (updates).

Encryption of voice data

Both parties *A* (the Caller) and *B* (the Callee) transform the voice information into a sequence of small *chunks* or *packets*, not more than 1 kilobyte each. This information is to be encrypted using the shared key *key* generated during the initial exchange, and sent to the other party, either directly (P2P) or through Telegram's relay servers (so-called *reflectors*). This document describes only the encryption process for each chunk, leaving out voice encoding and the network-dependent parts.

Encapsulation of low-level voice data

The low-level data chunk `raw_data:string`, obtained from voice encoder, is first encapsulated into one of the two constructors for the DecryptedDataBlock type, similar to DecryptedMessage used in secret chats:

```
decryptedDataBlock#dbf948c1 random_id:long random_bytes:string flags:# voice_call_id:flags.2?int128 in_seq_no:f
 simpleDataBlock#cc0d0e76 random_id:long random_bytes:string raw_data:string = DecryptedDataBlock;
```

Here `out_seq_no` is the chunk's sequence number among all sent by this party (starting from one), `in_seq_no` — the highest known out_seq_no from the received packets. The parameter `recent_received_mask` is a 32-bit mask, used to track delivery of the last 32 packets sent by the other party. The bit *i* is set if a packet with `out_seq_no` equal to `in_seq_no`-*i* has been received.

The higher 8 bits in `flags` are reserved for use by the lower-level protocol (the one which generates and interprets `raw_data`), and will never be used for future extensions of `decryptedDataBlock`.

The parameters `voice_call_id` and `proto` are mandatory until the other side confirms reception of at least one packet by sending a packet with a non-zero `in_seq_no`. After that, they become optional, and the `simpleDataBlock` constructor can be used if the lower level protocol wants to.

The parameter `voice_call_id` is computed from the key `key` and equals the lower 128 bits of its SHA-256.

The `random_bytes` string should contain at least 7 bytes of random data. The field `random_id` also contains 8 random bytes, which can be used as a unique packet identifier if necessary.

MTProto encryption

Once the data is encapsulated in `DecryptedDataBlock`, it is TL-serialized and encrypted with MTProto, using `key` instead of `auth_key`; the parameter *x* is to be set to *0* for messages from *A* to *B*, and to *8* for messages in the opposite direction. Encrypted data are prepended by the 128-bit `msg_key` (usual for MTProto); before that, either the 128-bit `voice_call_id` (if P2P is used) or the `peer_tag` (if reflectors are used) is prepended. The resulting data packet is sent by UDP either directly to the other party (if P2P is possible) or to the Telegram relay servers (reflectors).

Key Verification

To verify the key, both parties concatenate the secret key *key* with the value *g_a* of the Caller (*A*), compute SHA256 and use it to generate a sequence of emoticons. More precisely, the SHA256 hash is split into four 64-bit integers; each of them is divided by the total number of emoticons used (currently 333), and the remainder is used to select specific emoticons. The specifics of the protocol guarantee that comparing four emoticons out of a set of 333 is sufficient to prevent eavesdropping (MiTM attack on DH) with a probability of **0.9999999999**.

This is because instead of the standard Diffie-Hellman key exchange which requires only two messages between the parties:

- A->B : (generates a and) sends g_a := g^a
- B->A : (generates b and true key (g_a)^b, then) sends g_b := g^b
- A : computes key (g_b)^a

we use a **three-message modification** thereof that works well when both parties are online (which also happens to be a requirement for voice calls):

- A->B : (generates a and) sends g_a_hash := hash(g^a)
- B->A : (stores g_a_hash, generates b and) sends g_b := g^b
- A->B : (computes key (g_b)^a, then) sends g_a := g^a
- B : checks hash(g_a) == g_a_hash, then computes key (g_a)^b

The idea here is that *A* commits to a specific value of *a* (and of *g_a*) without disclosing it to *B*. *B* has to choose its value of *b* and *g_b* without knowing the true value of *g_a*, so that it cannot try different values of *b* to force the final key *(g_a)^b* to have any specific properties (such as fixed lower 32 bits of SHA256(key)). At this point, *B* commits to a specific value of *g_b* without knowing *g_a*. Then *A* has to send its value *g_a*; it cannot change it even though it knows *g_b* now, because the other party *B* would accept only a value of *g_a* that has a hash specified in the very first message of the exchange.

If some impostor is pretending to be either *A* or *B* and tries to perform a Man-in-the-Middle Attack on this Diffie—Hellman key exchange, the above still holds. Party *A* will generate a shared key with *B* — or whoever pretends to be *B* — without having a second chance to change its exponent *a* depending on the value *g_b* received from the other side; and the impostor will not have a chance to adapt his value of *b* depending on *g_a*, because it has to commit to a value of *g_b* before learning *g_a*. The same is valid for the key generation between the impostor and the party *B*.

The use of hash commitment in the DH exchange constrains the attacker to only **one guess** to generate the correct visualization in their attack, which means that using just over 33 bits of entropy represented by four emoji in the visualization is enough to make a successful attack highly improbable.

> For a slightly more user-friendly explanation of the above see: How are calls authenticated?